华章科技

中国FreeSWITCH领域“第一人”、全球FreeSWITCH开源社区知名专家撰写，FreeSWITCH之父鼎力推荐，权威性毋庸置疑

内容全面，涵盖FreeSWITCH使用、维护、二次开发、源码分析等各方面；实战性强，包含大量实例，从单个独立应用的实现到集群的部署，应有尽有

FreeSWITCH: The Definitive Guide

FreeSWITCH 权威指南

杜金房 张令考◎著

机械工业出版社
China Machine Press

VISIT…

LANZAROTE Caliente.com

FreeSWITCH
权威指南

FreeSWITCH: The Definitive Guide

FreeSWITCH是我2005年的一个梦想，到现在为止已经近10年了。它从那时的一个由5个文件组成的概念性的测试程序已经成长为现在的一个拥有50万行核心代码，另外还有其他300万行位于外部模块中的支持代码的软交换解决方案。我非常兴奋，因为我们的技术能够很快传达到13亿中国人那里，使得他们可以学习FreeSWITCH以及开源的电话技术。我希望这本书能够使FreeSWITCH中文社区更加壮大和繁荣。我鼓励本书的每位读者都加入在线的FreeSWITCH中文社区，与所有FreeSWITCH爱好者一起成长，并一起见证我们如何向着同一个目标前进，使该项目继续壮大、成长。

我在2008年第一次遇到Seven。从那时起一直到现在，他做了大量的工作——他竭尽所能学习了所有与FreeSWITCH相关的技术，并贡献了很多的Bug报告及修复代码。甚至他在为了写这本书而对FreeSWITCH进行研究时，还发现了至少5个Bug。没有他，我们可能永远也不能带给广大的中国读者如此好的一本理解FreeSWITCH的指导书。

—— **Anthony Minissale II**　FreeSWITCH之父

FreeSWITCH是VoIP的未来。它更加灵活、强大，并且与现有系统比起来更加节省成本。FreeSWITCH已稳步成为先进的VoIP系统部署的必备系统。本书的作者是中国FreeSWITCH界的权威，本书会指导你如何使用FreeSWITCH。杜金房曾建立了第一个中国的FreeSWITCH系统，安装了最的FreeSWITCH集群，并建立了FreeSWITCH中文社区。在中国的通信圈子里，没有人能比他更理解FreeSWITCH以及其内部的应用程序了。

—— **Jonathan Palley**　Spire公司CTO

上架指导：计算机/通信

ISBN 978-7-111-46626-0

定价：129.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991　88361066
购书热线：(010) 68326294　88379649　68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

FreeSWITCH: The Definitive Guide

FreeSWITCH 权威指南

杜金房 张令考◎著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

FreeSWITCH 权威指南 / 杜金房，张令考著．—北京：机械工业出版社，2014.5

ISBN 978-7-111-46626-0

I. F… II. ① 杜… ② 张… III. 电话交换 – 应用软件 – 指南 IV. TN916.4-62

中国版本图书馆 CIP 数据核字（2014）第 092466 号

FreeSWITCH 权威指南

杜金房 等著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：孙海亮 责任校对：殷 虹

印 刷：藁城市京瑞印刷有限公司 版 次：2014 年 6 月第 1 版第 1 次印刷

开 本：186mm × 240mm 1/16 印 张：40.25

书 号：ISBN 978-7-111-46626-0 定 价：129.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

前　言

我们已经步入了一个崭新的时代。当前，VoIP 已开始成为包括语音和视频在内的多媒体通信的主导并将在全世界范围内引领一场新的革命，而 SIP（Session Initiation Protocol，会话初始协议）正是这场革命的核心。

试想一下我们常用的电子邮件，它仅能用于文字通信。从电子邮件诞生开始，经过三十几年的发展，到现在几乎人人都有了一个 Email 地址（有的人还有多个）。随着实时多媒体通信需求的增加及技术的发展，笔者相信在不久的将来，每个人都将会拥有一个用于多媒体通信的 SIP 地址。当前，因为互联网技术高速发展，使得数据流量的成本越来越低，而且随着 3G、4G 无线网络的不断发展与优化，以及各种智能移动终端的出现，网络更将无处不在，这就为 SIP 的进一步普及营造了必要的环境。在这种环境下，各种新型的 SIP 电话及可以运行在各种移动设备上的 SIP 客户端可以让用户以极低的成本与世界上任何一个角落的人通信。

为什么写这本书

大多数关于操作系统的图书均重理论而轻实践，而本书则在这两者之间进行了较好的折中。

—— Andrew S. Tanenbaum

从第一次读 Tanenbaum 的《操作系统设计与实现》到现在已经十多年了，可这句写在那本书前言里的话到现在还让我记忆犹新。上大学时，我也曾学习过“程控交换网”、“移动通信”之类的专业知识，但那时只有肤浅的认识，理解不深。毕业后，我到烟台电信[⊖]工作，负责程控交换机的维护。在工作中，我学到了电信网络传输与交换的各种技术，掌握了七号信令系统（SS7），算是做到了理论与实践相结合吧。那时候，VoIP 还是很新的东西，由于网络条件的限制，国内也少有人用。2007 年年底，我开始接触 Asterisk。阅读了《Asterisk，电话未来之路》，并买了一个单口的语音卡，实现

⊖ 这个名字也许太老了。中国电信业在短短的几年内经过了数次重组改制，我离开时叫烟台网通，烟台电信是我刚参加工作时的名字，与现在的烟台电信不是同一家公司。

了 VoiceMail、PSTN 网关、SIP 中继等各种功能。能在自己电脑上实现这些有趣的东西，令我非常兴奋。后来，我加入 Idapted Inc.，开发一种一对一的网络教学平台。最初的后台语音交换系统也使用 Asterisk，但不久后便转到了 FreeSWITCH。那时我就深切意识到，FreeSWITCH 必将成为 Asterisk 之后的通信领域的又一宠儿。于是便怀着无比的热情开始学习 FreeSWITCH。

在学习和探索 FreeSWITCH 的过程中，我逐渐融入了 FreeSWITCH 社区，与全世界的开发者讨论各种 VoIP 技术、交流使用经验，这让我学到了许多知识。同时，在开发和使用 FreeSWITCH 的过程中我也积累了大量的使用和开发经验，并将这些知识和经验不时地更新到 FreeSWITCH 官方的 Wiki 上。Wiki 是 FreeSWITH 文档的大本营，内容来自众多 FreeSWITCH 爱好者和实践者的奉献。不过，对于初学者来说，Wiki 上的内容读起来还不是很系统，因此我就萌生了写一本书的想法。

本书面向的读者

本书作为国内第一本介绍 FreeSWITCH 的书，内容全面又不失深度，因此适合从事 FreeSWITCH 相关工作的各层次的读者阅读——对于初学者来说，本书是很好的入门教材；对于有一定经验的读者来说，本书可以提供更深入的指导和案例分析，帮助他们深入理清各功能点的来龙去脉；对于高级用户和开发者来说，本书也有详细的开发指导和最佳实战经验。

- **FreeSWITCH 爱好者**：对于 FreeSWITCH 爱好者来说，本书是最直接的需求。本书内容全面、介绍细腻和深度适当，是其他资料或同类书籍所没有的。
- **以 FreeSWITCH 为基础进行二次开发的人员**：以 FreeSWITCH 为基础，可以进行呼叫中心、指挥调度、IP-PBX、互联网音频或视频等系统的开发。本书可以提供详尽、丰富的参考资料和实践经验，使这类读者少走弯路，甚至达到事半功倍的效果。
- **VoIP 系统、软交换系统、电信设备开发人员**：FreeSWITCH 是开源领域的典范，又是同类系统中的佼佼者。本书对 FreeSWITCH 的原理、实现都做了深入探讨，还对源代码进行了导读和深入分析，可以带领读者很快了解现代软件的架构、逻辑及算法实现，因而很有借鉴意义。FreeSWITCH 使用 MPL 许可证发布，完全不限制商业应用，因而广大开发者也可以完全站在巨人肩膀上，不重复发明轮子，迅速做出新的产品。
- **Asterisk 开发者**：跟大多数 Asterisk 开发者一样，我也是读着《Asterisk，电话未来之路》、《Trixbox[⊖]不相信眼泪》一路走过来的。FreeSWITCH 作为后起之秀，在许可证、系统架构和性能方面都有很好的优势。了解 FreeSWITCH 可以开拓读者的眼界和思路，不管是否转到 FreeSWITCH，相信本书都会对你的职业生涯有所帮助。
- **FreeSWITCH 系统实施、维护人员**：对于 FreeSWITCH 相关呼叫中心、IP-PBX 等系统的

⊖ 基于 Asterisk 的一款 PBX 软件。

实施和维护人员来说，本书也是不可或缺的教材。事实上，本书第 3 章介绍了一个完整的 FreeSWITCH IP-PBX 搭建方案，通过对这一章的学习，企业的系统管理员可以很容易地用 FreeSWITCH 打造一个企业电话系统。

- **电信企业的维护人员、技术决策人员**：广大电信企业的人员在以往的工作中积累了大量的工作经验，但往往依赖于华为、中兴等设备厂家提供的解决方案和技术架构。技术瞬息万变，在市场竞争日益激烈，国内电信政策调整并逐渐宽松之际（如虚拟运营商牌照的发放），只有了解另一种解题思路，并掌握了新技术，才能更好地把握市场方向，为客户提供更好的服务。
- **呼叫中心从业人员**：可以预见，在不远的将来，将有很多呼叫中心是基于 FreeSWITCH 开发的。而本书中丰富的基础知识和详尽的功能介绍将对呼叫中心系统的使用、管理和决策起到很好的指导作用。
- **相关专业在校师生**：笔者看过一些学校的教材，大部分只是讲 VoIP 原理及 SIP 协议等，很枯燥；而且，老师总是教育学生"要理论与实践相结合"，而本书正是理论与实战的最佳结合点。同时我也注意到，有一些学校的导师和研究生已经在围绕这个主题进行研究了。
- **与 FreeSWITCH 相关的开发经理、技术决策人员**：FreeSWITCH 是 VoIP 软交换软件的典范，但本书所讲的不仅仅是 FreeSWITCH。书中以 FreeSWITCH 为例将枯燥的（包括通信、互联网以及软件开发）专业知识以浅显、直观、生动的语言呈现在读者面前；同时，对软件开发的模型、模式、架构、流程、状态机、接口设计等进行了普及和深入剖析。了解本书所讲的知识有助于技术选型和决策。

总之，FreeSWITCH 是一个电话系统。我们人人都离不开电话，可以说，它将来可能出现在任何企业、部门及要使用电话的所有角落。进一步讲，FreeSWITCH 完全超越了一个简单的电话系统，其应是一个更通用的通信系统平台，它可以完美地与互联网及移动互联网结合，创建前所未有的通信应用。古人云"开卷有益"，或云"他山之石，可以攻玉"。所以，即使你不使用 FreeSWITCH，但只要从事与通信相关的工作，本书都会在扩充基础知识、开阔思路等方面对你有所帮助。

本书的内容及特色

作为专门介绍 FreeSWITCH 的第一本中文书，笔者对内容做了精心安排。为了照顾各种背景、各种水平的读者，也为了让读者更多地了解 FreeSWITCH 的方方面面，本书尽量选择了不同语言、不同场景、不同技术、不同实现方法的例子，力求使大家通过本书，对 FreeSWITCH 有一个全面而又深入的认识。

本书内容基于最新的 FreeSWITCH 1.4 版（功能和特性截止到 2014 年 1 月份的最新开发版本），从总体上分为三个部分：

第一部分　基础篇：作为本书的开篇，这部分首先介绍一些基本概念及背景知识，这部分内容主

要针对没有电信背景的人。对从电路交换转到 VoIP 来的读者来说也会很有帮助。此外，这部分也详尽讲解了学习 FreeSWITCH 应该具备的基础知识，如基本的概念和术语、FreeSWITCH 的基本架构、模块的组织、配置文件的组织、拨号计划和电话路由、SIP 协议、媒体处理与媒体协商、NAT 穿越，以及 FreeSWITCH 中的 SIP 模块（mod_sofia）等，并在第 3 章带领大家进行了完整的 IP-PBX 配置和体验。

读完本部分，读者就能具备独立使用和维护基本的 FreeSWITCH 系统的能力了。

第二部分　实战篇：要想学好 FreeSWITCH，就必须做到理论与实践相结合，所以反复练习是必须的。这部分结合我在实际工作中的一些实际例子，介绍了在解决问题的过程中积累的经验。如第 10 章介绍了调试与排错的一般技巧、检查及分析系统日志的方法，并详细解释了呼叫是如何工作的。在后面的章节中还讲了一些基本功能的配置实例，如创建用户、IVR、录放音、TTS、呼叫排队、视频通话、多人电话会议、话单、计费、FreeSWITCH 对接以及生产环境中的应用、系统的部署方式、拓扑结构、参数配置、安全性考虑及系统优化指南等。

通过这些内容，读者可以深入理解基础篇介绍的概念和理论，并且可以仿照这里的例子做出各种各样的应用。另外，这部分中有些例子甚至可以直接套用。

第三部分　高级篇：这部分主是面对开发者的。不过，即使不从事开发工作，学习一下本部分的内容也是很有帮助的。这部分除了讲开发之外，还对整个系统架构及内部实现进行了深入剖析。读完后你会觉得：噢，原来是这样的！

具体来讲，这部分讲了嵌入式脚本、Event Socket 开发的基础知识以及通过这些开发技术实现的 IVR、呼叫中心应用之类的各种例子。此外，还讲解了源代码阅读、编译的方法，并对源代码进行了深入分析。最后给出了一个在 FreeSWITCH 代码上进行源代码级二次开发的实例。读者可以通过自己动手实现各种呼叫功能和流程，进一步理解和巩固所学的知识，以便从根本上了解 FreeSWITCH 中的一切秘密。

排版及约定

- 本书部分插图由 XMind、Keynote、Graphviz、mscgen 等工具生成。
- 程序代码、系统的输入输出等大部分有行号。强调部分使用粗体字。
- 有些程序代码行较长，为适应版面，进行了人工排版。
- 提示符。对于命令行的输入输出来说，在 Linux 及 Mac 等 UNIX 类平台上，前面的“$”（或“#”，如果为 root 用户的话）符为操作系统命令提示符；在 Windows 系统上使用“C:\”作为操作系统命令提示符；“freeswitch>”作为 FreeSWITCH（控制台以及 fs_cli）系统提示符。在不至于引起混淆的情况下，可能会省略系统提示符。
- 除非有特别说明，本书中的 Wiki 一般指 wiki.freeswitch.org，Jira 一般指 jira.freeswitch.org。

如何阅读本书

本书对内容和学习路线做了精心安排，对于大部分读者而言，适合顺序阅读（唯一例外的就是你可以先读后记部分）。当然，由于各种概念和理论都是相互联系的，因此，在读后面一些章节时回过头来复习和印证一下某些基本概念也是很有必要的。

对于刚进入通信领域的开发者来说，前面的通信历史和基础知识是非常有用的。只有了解了这些基础知识，才能更好地理解通信网中各个不同组成部分所发挥的作用以及它们之间的交互流程。对于有了通信基础却不熟悉 FreeSWITCH 的读者来说，可以简单浏览前面的通信基础知识，并通过第 3 章的 FreeSWITCH 体验对 FreeSWITCH 有一个直观的认识。对于比较熟悉 FreeSWITCH 的读者来说，本书的第 4 章到第 9 章也会帮助他们使以前零散的知识更加系统化。对于基于 FreeSWITCH 进行开发的普通开发者来说，在熟练掌握基础知识之后，再阅读本书第二部分，并同步做一些练习，会取得较好的效果。如果想直接在 FreeSWITCH 内部开发模块或修改 FreeSWITCH 的源代码，可接着顺序阅读第三部分的内容。

当然，笔者鼓励所有人都通读全书的内容，即使不做开发，后面有关开发的相关内容也有助于理解前面的基础知识。在通读本书一遍以后，根据自己的情况挑选相关的章节进行反复阅读和实验，以便将所有知识都融会贯通，从而使自己的水平更上一层楼。

此外，FreeSWITCH 一直处于很活跃的开发中，所以某些章节中描述的内容可能与最新的版本有所出入，我将在本书相关的网站上进行跟踪说明。

实验环境及软、硬件设备选择

本书涉及的软件绝大部分都是开源软件（非开源的软件，笔者都进行了特殊说明），因此读者可以根据书中的指导自行安装和实验。

为了使书中的案例更真实，同时也为了方便大家在真实的硬件环境中进行实践，书中的实验我们均使用真实的硬件设备或软件产品进行。

作为一本技术书籍，我们在书中只讨论各种设备互连的技术性问题，而对于各厂商的设备好坏则不做评论。有条件的读者可以自行研究和比较。不过，如果读者在学习过程中想拿硬件设备做实验，但是不知道该选择哪款产品，我建议可以从本书中提到的设备中选择。推荐的理由有二：其一，这些设备我都亲自测试过；其二，书中的硬件都是现在市场上比较流行、口碑和质量都比较好的，这类硬件不容易出错，即使出错也容易找到技术支持。初学者往往对 FreeSWITCH 和硬件都不是很熟悉，因而在遇到问题时很难定位是硬件原因还是软件原因，以至于浪费宝贵的学习时间。至于在生产环境中大规模的应用，相信各公司都有专门的人研究，我在此就不敢妄言了。

资源和勘误

- book.dujinfang.com 是本书的在线站点，提供本书的代码下载及勘误等。
- freeswitch@dujinfang.com 是专门为本书设立的电子邮箱，如果您对本书有任何意见、建议或批评请发到该邮箱。
- http://weibo.com/dujinfang 是我的微博，我很乐意与您进行交流。
- FreeSWITCH-CN 是 FreeSWITCH 中文社区的微信公共账号，您也可以关注该账号获取本书最新的情况，以及与本书作者进行交流。
- www.freeswitch.org.cn 是 FreeSWITCH-CN 中文社区的官方网站，经常关注该网站的更新，这样可以获取中文社区的最新信息。
- www.hzbook.com 是华章公司的官网，您也可以到这里下载相关代码。

由于笔者水平所限，书中存在错误和疏漏在所难免，欢迎广大读者批评指正。

致谢

FreeSWITCH™ 是 OSTAG ⊖的注册商标。感谢 Anthony Minessale 及他的团队给我们提供了如此优秀的软件，也感谢他在本书写作时提供的耐心指导和答疑；同时感谢 FreeSWITCH 社区所有成员的热心帮助，使我在学习 FreeSWITCH 的道路上一帆风顺。本书的一些资料和例子来自 FreeSWITCH Wiki 及邮件列表，因不能一一查证原作者，故在此一并致谢。

感谢机械工业出版社华章公司。华章公司对中文原创计算机图书的信任让我倍感温馨。感谢杨福川策划，他对图书出版的理念带给我许多启发和写作的动力。感谢孙海亮和姜影编辑，他们的耐心和细致保证了本书的质量和写作进度。感谢方舟先生，他给了我许多写书的建议，并介绍我认识杨福川先生。感谢刘光大老师，是他帮我策划并将我带上了出版之路。

感谢我的妻子吕佳娉，她是本书的第一读者，常常帮我修订文字错误；偶尔，在我埋头写作时，悄悄地在我案头放一个削好的苹果；多少次，在我挑灯夜战的翌日清晨，做好早餐，冲好茶，自己悄悄去上班。感谢我的儿子杜昱凝，他很聪明，还不到 4 岁就会经常帮我测试电话了。在我写作时，他清脆的声音“喂，爸爸吗？……”经常萦绕在我的耳畔，鼓励我继续写下去。我也很内疚写书占用了好多本应该陪他玩耍的时间。

感谢 Jonathan Palley 先生，是他指导我走上了 FreeSWITCH 之路。感谢崔钢先生跟我一起创业，因为有了他管理公司的方方面面，才使我有更多的时间专注于 FreeSWITCH ；此外，他还写了本书第 2 章中有关呼叫中心的部分内容。感谢滨州公路工程总公司机械工程师张令考先生，他与笔者合作编

⊖ Open Source Telephony Advancement Group，即开源电话学进步组，参见 http://ostag.org/。

写了第 2 章、第 12 章及第 14 章的部分内容。感谢高超、李洋、杨小金、景朝阳、李雪梅，他们或者写了一些例子，或者帮我进行了校对、测试、编辑和排版等。感谢程祝波，他维护着 FreeSWITCH-CN QQ 群，从而为大家提供了一个实时交流的平台，同时他也对本书的写作提了很多很好的意见。感谢 Tim Yang 先生，是他提供了本书自写作以来的第一笔捐款。感谢网友 flyingnn 及其他热心读者，他们对本书给予了积极的反馈并帮助我勘误，没有他们，便没有本书。

特别鸣谢

在本书写作时，得到了国内外一些硬件和设备厂商的赞助和支持，在此表示由衷的感谢。这些公司有（排名不分先后）：Yealink（亿联）、Grandstream（潮流）、Dinstar（鼎信通达）、Sangoma（加拿大，及中国总代星昊通）、Vestec（加拿大）等。

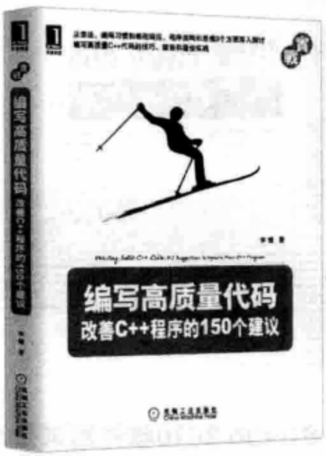
杜金房

推荐阅读

从语法、应用架构、工具、框架、编码风格和编程思想等多角度探讨编写高质量代码的最佳实践，为进阶修炼提供绝佳指导！

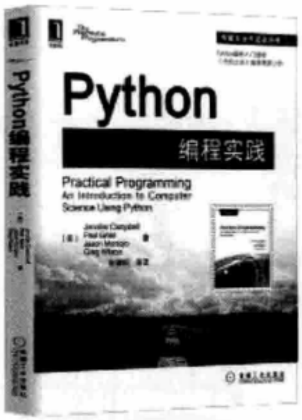






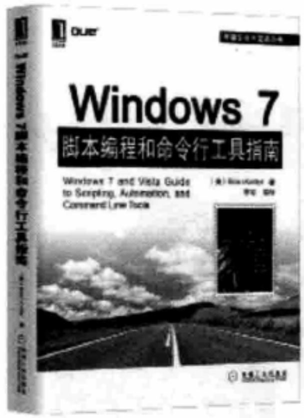


推荐阅读



Python编程实践

作者：Jennifer Campbell ISBN：978-7-111-36478-8 定价：49.00元



Windows 7脚本编程和命令行工具指南

作者：Brian Knittel ISBN：978-7-111-35677-6 定价：79.00元



Linux内核设计与实现（原书第3版）

作者：Robert Love ISBN：978-7-111-33829-1 定价：69.00元



Visual Studio 2010软件测试指南

作者：Jeff Levinson ISBN：978-7-111-35931-9 定价：49.00元

目　　录

前言

第一部分　基　础　篇

第 1 章　PSTN 与 VoIP 基础 ……2
1.1　PSTN 起源与发展 ……2
1.1.1　最早的电话网 ……2
1.1.2　人工电话交换时代 ……3
1.1.3　自动电话交换时代 ……3
1.1.4　半电子交换机时代 ……6
1.1.5　空分交换机时代 ……6
1.1.6　数字交换机时代 ……6
1.1.7　现代 PSTN 时代 ……6
1.1.8　下一代网络及 VoIP 时代 ……6
1.2　电话实现技术 ……8
1.2.1　电话号码 ……8
1.2.2　模拟信号与数字信号 ……11
1.2.3　PCM ……11
1.2.4　局间中继与电路复用技术 ……12
1.3　我国电话网结构 ……12
1.4　信令 ……14
1.4.1　信令分类 ……14

1.4.2 用户线信令 ……15
1.4.3 局间信令 ……15
1.4.4 七号信令 ……16
1.4.5 H.323 与 SIP 信令 ……17
1.5 媒体 ……19
1.6 电路交换与分组交换 ……20
1.6.1 电路交换 ……20
1.6.2 分组交换 ……20
1.7 VoIP ……21
1.8 IMS ……22
1.8.1 什么是 IMS ……22
1.8.2 IMS 的特点 ……22
1.8.3 IMS 核心网元 ……23
1.8.4 SIP 协议的参考点 ……26
1.9 小结 ……27

第 2 章 PSTN、PBX 及呼叫中心业务 ……28

2.1 PSTN 业务 ……28
2.1.1 POTS ……28
2.1.2 商务业务 ……30
2.1.3 其他增值业务 ……30
2.2 PBX 业务 ……31
2.2.1 呼叫转移 ……31
2.2.2 同组代答 ……31
2.3 PBX 与中继线 ……31
2.4 IP-PBX 业务 ……34
2.5 呼叫中心 ……35
2.5.1 什么是呼叫中心 ……35
2.5.2 呼叫中心的历史 ……36
2.5.3 呼叫中心的分类 ……37
2.5.4 呼叫中心的主要技术指标 ……39
2.5.5 CTI 中间件 ……40

2.5.6 FreeSWITCH 在呼叫中心的应用 …… 41
2.6 小结 …… 42

第 3 章 初识 FreeSWITCH …… 43

3.1 什么是 FreeSWITCH？ …… 43
3.1.1 FreeSWITCH 的概念 …… 43
3.1.2 FreeSWITCH 的功能 …… 45
3.2 快速体验 …… 46
3.2.1 安装基本 FreeSWITCH 系统 …… 46
3.2.2 连接 SIP 电话 …… 57
3.3 配置 FreeSWITCH …… 61
3.4 FreeSWITCH 用作软电话 …… 63
3.5 配置 SIP 网关拨打外部电话 …… 64
3.5.1 从某一分机上呼出 …… 65
3.5.2 呼入电话处理 …… 66
3.6 小结 …… 66

第 4 章 运行 FreeSWITCH …… 67

4.1 命令行参数 …… 67
4.2 系统启动脚本 …… 68
4.3 判断 FreeSWITCH 是否运行 …… 69
4.4 控制台与命令客户端 …… 69
4.5 呼叫 …… 72
4.5.1 发起呼叫 …… 72
4.5.2 呼叫字符串 …… 72
4.6 API 与 App …… 73
4.7 API 命令帮助 …… 75
4.8 小结 …… 76

第 5 章 FreeSWITCH 架构 …… 77

5.1 总体架构 …… 77
5.1.1 核心 …… 77

5.1.2 接口实现 ······81
5.2 目录结构 ······83
5.3 配置文件 ······85
5.3.1 freeswitch.xml ······86
5.3.2 vars.xml ······88
5.3.3 autoload_configs 目录 ······89
5.3.4 其他 ······90
5.4 XML 用户目录 ······91
5.5 呼叫相关概念 ······93
5.5.1 来去话、Session、Channel 与 Call ······93
5.5.2 回铃音与 Early Media ······94
5.5.3 全局变量与局部变量 ······94
5.6 小结 ······95

第 6 章 拨号计划 ······96

6.1 XML Dialplan ······96
6.1.1 配置文件的结构 ······96
6.1.2 默认的配置文件简介 ······99
6.1.3 正则表达式 ······100
6.1.4 通道变量 ······101
6.1.5 测试条件 ······105
6.1.6 动作与反动作 ······108
6.1.7 工作机制深入剖析 ······109
6.1.8 内联执行 ······111
6.1.9 实例解析 ······112
6.2 inline Dialplan ······118
6.3 其他 Dialplan ······120
6.4 常用的 Dialplan App ······120
6.5 在 Dialplan 中使用 API 命令 ······123
6.6 深入理解通道变量及相关操作 ······124
6.7 小结 ······125

第 7 章 SIP 协议 …… 126
7.1 SIP 协议基础 …… 126
7.1.1 HTTP 与 SIP 协议基础 …… 127
7.1.2 SIP 的基本概念和相关元素 …… 129
7.1.3 SIP 协议的基本方法和头域简介 …… 131
7.2 SIP 注册 …… 132
7.3 SIP 呼叫流程 …… 135
7.3.1 UA 间直接呼叫 …… 136
7.3.2 通过 B2BUA 呼叫 …… 137
7.4 深入理解 SIP …… 147
7.4.1 SIP URI …… 147
7.4.2 SDP 和 SOA …… 147
7.4.3 3PCC …… 151
7.4.4 SIP 承载 …… 152
7.5 小结 …… 152

第 8 章 媒体 …… 154
8.1 媒体与媒体处理 …… 154
8.1.1 音频编码 …… 154
8.1.2 媒体工作机理和相关配置 …… 159
8.2 媒体协商 …… 160
8.2.1 协商过程 …… 160
8.2.2 SDP 及其在编码协商中的作用 …… 161
8.2.3 协商时机与策略 …… 163
8.3 其他媒体相关的问题 …… 163
8.3.1 RTP 和 RTCP …… 163
8.3.2 转码 …… 165
8.3.3 透传、媒体绕过与媒体代理 …… 167
8.3.4 Media Bug …… 168
8.3.5 视频 …… 168
8.3.6 排错 …… 169
8.4 小结 …… 170

第 9 章　SIP 模块……171

9.1　基本概念……171
9.2　Sofia 配置文件……172
9.2.1　Profile 配置文件……173
9.2.1　Profile 的几个重要参数……175
9.2.2　external.xml……177
9.2.3　Gateway……179
9.3　常用命令……181
9.3.1　状态相关命令……181
9.3.2　Profile 相关命令……182
9.3.3　SIP Capture……183
9.3.4　global 相关……184
9.3.5　debug 相关……184
9.3.6　其他命令……185
9.3.7　其他……186
9.4　NAT 穿越……186
9.4.1　NAT 的种类……188
9.4.2　FreeSWITCH 的拓扑结构……189
9.4.3　NAT 是怎么影响 SIP/RTP 通信的……190
9.4.4　NAT 的穿越方法……191
9.5　小结……198

第二部分　实　战　篇

第 10 章　基本技能……200

10.1　调试与排错……200
10.1.1　解决问题的一般方法和流程……200
10.1.2　查看日志……203
10.2　使用外部工具抓包……205
10.2.1　tcpdump……205
10.2.2　tshark……206

10.2.3 ngrep ······ 207
10.2.4 pcapsipdump ······ 208
10.3 使用 Wireshark 抓包并分析呼叫 ······ 208
10.3.1 使用 Wireshark 抓包 ······ 209
10.3.2 使用 Wireshark 对抓包进行分析 ······ 210
10.4 originate 命令实例解析 ······ 213
10.4.1 使用格式和参数 ······ 213
10.4.2 转入 Dialplan ······ 214
10.4.3 更改主叫号码 ······ 215
10.4.4 处理呼叫超时 ······ 215
10.4.5 防止命令阻塞 ······ 216
10.4.6 使用通道变量 ······ 216
10.4.7 Early Media 对呼叫的影响 ······ 217
10.4.8 bridge 也使用 originate ······ 217
10.4.9 bridge 中的 Early Media ······ 218
10.4.10 bridge 中的主叫号码 ······ 219
10.5 呼叫是怎样工作的？ ······ 220
10.6 FreeSWITCH 图形用户界面简介 ······ 223
10.6.1 FusionPBX ······ 223
10.6.2 blue.box ······ 223
10.6.3 FreeSWITCH Portal ······ 224
10.7 小结 ······ 226

第 11 章 基本功能与实现 ······ 227

11.1 批量创建用户 ······ 227
11.2 用 FreeSWITCH 实现 IVR ······ 229
11.2.1 最简单的菜单 ······ 229
11.2.2 默认 IVR 简介 ······ 232
11.3 按时间进行路由 ······ 233
11.4 配置中文语音提示 ······ 234
11.4.1 最简单的实现方案 ······ 234
11.4.2 使用 sound_prefix ······ 235

11.4.3 使用 Phrase …… 236
11.4.4 使用中文语音提示 …… 238
11.5 录音 …… 239
11.5.1 单腿录音 …… 239
11.5.2 对两条腿的通话进行录音 …… 240
11.5.3 立体声 …… 241
11.5.4 录音相关的通道变量 …… 242
11.5.5 原生格式 …… 243
11.6 放音 …… 244
11.6.1 playback 的参数 …… 244
11.6.2 循环播放 …… 248
11.6.3 Say …… 249
11.7 TTS …… 251
11.7.1 使用 mod_flite …… 251
11.7.2 mod_tts_commandline …… 252
11.7.3 MRCP …… 256
11.7.4 Google Translate …… 257
11.7.5 TTS 小结 …… 258
11.8 在呼叫失败的情况下向主叫用户播放语音提示 …… 259
11.8.1 实现方法 …… 259
11.8.2 进阶 …… 260
11.8.3 使用 TTS …… 261
11.8.4 原理 …… 262
11.9 实现呼叫前转业务 …… 263
11.10 小结 …… 265

第 12 章 高级功能与配置实例 …… 266

12.1 使用 mod_fifo 实现简单呼叫队列 …… 266
12.1.1 呼叫停泊与取回 …… 266
12.1.2 配置座席 …… 268
12.1.3 fifo …… 270
12.2 使用 mod_callcenter 实现呼叫中心应用 …… 272

12.2.1 mod_callcenter 模块简介 ······ 273
12.2.2 座席配置与管理 ······ 274
12.3 数据库 ······ 277
12.3.1 默认数据库中有什么? ······ 277
12.3.2 ODBC ······ 279
12.3.3 使用数据库原生客户端库直接连接数据库 ······ 282
12.4 视频通话 ······ 283
12.4.1 配置视频通话 ······ 283
12.4.2 视频录像与回放 ······ 285
12.4.3 视频转码 ······ 285
12.5 多人电话会议 ······ 286
12.5.1 音频会议 ······ 286
12.5.2 视频会议 ······ 292
12.6 话单 ······ 297
12.6.1 CSV 格式的话单 ······ 297
12.6.2 直接将话单写入数据库 ······ 298
12.6.3 使用 HTTP 服务器接收话单 ······ 299
12.7 计费 ······ 300
12.8 小结 ······ 302

第 13 章 FreeSWITCH 与 FreeSWITCH 对接 ······ 303

13.1 在同一台主机上启动多个 FreeSWITCH 实例 ······ 303
13.1.1 背景故事 ······ 304
13.1.2 练习 ······ 305
13.1.3 进阶 ······ 306
13.2 FreeSWITCH 与 FreeSWITCH 对接 ······ 306
13.2.1 双机对接 ······ 306
13.2.2 汇接 ······ 308
13.2.3 双归属 ······ 309
13.2.4 长途局 ······ 311
13.2.5 ACL ······ 311
13.3 FreeSWITCH 作为 PBX ······ 312

13.3.1 普通的 PBX 设置 ……313
13.3.2 DID……315
13.3.3 使用 PBX 上的网关呼出 ……319
13.4 小结 ……319

第 14 章 FreeSWITCH 与其他设备或系统对接 ……321

14.1 使用 Doubango 客户端连接 ……321
14.2 对接 IMS ……322
14.2.1 网关配置 ……323
14.2.2 通过 IMS 呼出……324
14.2.3 通过 IMS 呼入……324
14.2.4 其他问题 ……325
14.3 连接模拟话机和模拟中继线 ……326
14.3.1 FXS 和 FXO ……326
14.3.2 拓扑结构 ……327
14.3.3 使用潮流网关连接模拟话机……328
14.3.4 使用迅时网关连接模拟话机和模拟中继线 ……329
14.4 通过 E1 线路与其他系统对接……335
14.4.1 配置 FS1……335
14.4.2 配置 E1 网关设备……340
14.4.3 配置 FS2……342
14.4.4 对接其他厂家的 E1 网关 ……345
14.5 对接 Asterisk ……346
14.5.1 从 FreeSWITCH 呼叫 Asterisk……346
14.5.2 从 Asterisk 上呼叫 FreeSWITCH……347
14.5.3 其他……347
14.6 使用 H.323 协议对接……348
14.6.1 mod_h323……348
14.6.2 mod_opal……350
14.6.3 其他……351
14.7 小结 ……351

第 15 章　其他技巧与实例 ………… 353

15.1　转接和代接 ………… 353

15.1.1　盲转 ………… 353

15.1.2　协商转 ………… 356

15.1.3　代接 ………… 357

15.2　共享线路呈现 ………… 358

15.3　使用组播功能做网络广播 ………… 360

15.4　DTMF ………… 361

15.4.1　带内 DTMF ………… 361

15.4.2　RFC2833 ………… 362

15.4.3　SIP INFO ………… 363

15.5　号码连选 ………… 363

15.5.1　注册到运营商服务器 ………… 364

15.5.2　通过单个号码呼出 ………… 364

15.5.3　使用随机数做号码连选 ………… 365

15.5.4　使用 mod_distributor 进行连选 ………… 365

15.5.5　其他 ………… 366

15.6　收发传真 ………… 367

15.7　多租户 ………… 368

15.7.1　Domain 简介 ………… 368

15.7.2　配置与实例 ………… 370

15.7.3　进阶 ………… 372

15.7.4　其他 ………… 373

15.8　使用 loopback Endpoint 外呼 ………… 373

15.9　在 Web 浏览器中打电话 ………… 375

15.9.1　Flash ………… 376

15.9.2　WebRTC ………… 379

15.10　HA ………… 382

15.10.1　崩溃恢复实验 ………… 382

15.10.2　HA 简介 ………… 383

15.10.3　双机 HA 实现细节及需要注意的问题 ………… 384

15.11　集群及分布式部署 ………… 386

15.11.1 大规模集群的总体结构……386
15.11.2 负载均衡配置实例……387
15.12 压力测试……392
15.12.1 参数和指标……392
15.12.2 呼叫测试……393
15.12.3 注册测试……395
15.12.4 编解码测试……396
15.12.5 测试结果……396
15.13 生产环境下的稳定性和安全性……396
15.13.1 稳定性……397
15.13.2 安全性……397
15.14 小结……398

第三部分 高 级 篇

第 16 章 嵌入式脚本……400
16.1 FreeSWITCH 中的嵌入式脚本……400
16.2 Lua……401
16.2.1 Lua 语法简介……402
16.2.2 将电话路由到 Lua 脚本……408
16.2.3 Session 相关函数……408
16.2.4 非 Session 函数和独立的 Lua 脚本……412
16.2.5 Event 相关函数……412
16.2.6 Chat 相关函数……415
16.2.7 LUA 拨号计划……416
16.2.8 连接数据库……417
16.3 其他脚本语言……419
16.3.1 JavaScript……419
16.3.2 Python……420
16.4 小结……421

第 17 章 嵌入式及 HTTP 开发 …… 423
17.1 用 Lua 脚本写个小游戏 …… 423
17.2 用 Lua 实现 IVR …… 425
17.3 在会议中呼出 …… 429
17.4 一个在 FreeSWITCH 中外呼的脚本 …… 431
17.5 使用 Lua 脚本通过多个网关循环外呼 …… 433
17.6 在 FreeSWITCH 中执行长期运行的嵌入式脚本 …… 434
17.7 使用 Lua 提供 XML Binding …… 436
17.8 语音识别 …… 437
17.8.1 使用 Pocket Sphinx 进行中文语音识别 …… 438
17.8.2 通过商业语音识别软件进行识别 …… 444
17.9 使用 mod_xml_curl 提供动态用户管理 …… 446
17.10 使用 mod_xml_cdr 模块处理话单 …… 448
17.11 小结 …… 450

第 18 章 Event Socket …… 452
18.1 架构 …… 452
18.1.1 外连模式 …… 453
18.1.2 内连模式 …… 454
18.2 Event Socket 协议 …… 454
18.2.1 外连 …… 455
18.2.2 内连 …… 456
18.2.3 Event Socket 命令详解 …… 457
18.3 Event Socket 库 …… 465
18.3.1 Event Socket 示例 …… 465
18.3.2 ESL 函数说明 …… 470
18.4 事件 …… 475
18.4.1 事件的学习方法 …… 475
18.4.2 常用事件简介 …… 476
18.5 小结 …… 478

第 19 章　使用 ESL 开发 …… 479

19.1　创建独立的 ESL 应用 …… 479

19.1.1　创建目录和源文件 …… 479

19.1.2　编译和执行 …… 480

19.1.3　将源代码纳入版本控制 …… 481

19.2　用 ESL 重写空中充值服务 …… 482

19.3　用 ESL 写一个 ACD …… 490

19.4　用 Inbound 模式实现 IVR …… 496

19.5　使用 Java 连接 ESL …… 501

19.6　使用 Erlang 控制呼叫流程 …… 502

19.6.1　准备工作 …… 503

19.6.2　将来话交给 Erlang 处理 …… 504

19.6.3　用 Erlang 实现空中充值服务 …… 506

19.6.4　用 Erlang 状态机实现空中充值服务 …… 509

19.6.5　其他 …… 514

19.7　定时呼叫 …… 516

19.8　小结 …… 517

第 20 章　源代码导读及编译指南 …… 518

20.1　准备 FreeSWITCH 源代码环境 …… 518

20.2　FreeSWITCH 源代码目录结构 …… 519

20.3　FreeSWITCH 源代码导读 …… 519

20.3.1　APR …… 520

20.3.2　SWITCH APR …… 520

20.3.3　main 函数 …… 522

20.3.4　可加载模块 …… 524

20.3.5　模块的结构 …… 530

20.3.6　Session 和 Channel …… 531

20.3.7　SWITCH IVR …… 536

20.3.8　Core IO …… 537

20.3.9　Core Media …… 540

20.3.10　Core RTP …… 541

20.3.11 SWITCH XML······543
20.3.12 SWITCH Event······544
20.3.13 Core Codec 和 Core File······547
20.4 FreeSWITCH 源代码的编译······548
20.4.1 首次编译······549
20.4.2 增量编译······550
20.4.3 常见问题及最佳实践······551
20.5 小结······552

第 21 章 FreeSWITCH 源代码分析······553
21.1 mod_dptools······553
21.1.1 echo······553
21.1.2 answer······555
21.1.3 set······557
21.1.4 bridge······559
21.1.5 Endpoint Interface······563
21.1.6 模块框架······564
21.2 mod_commands······566
21.2.1 模块框架······566
21.2.2 originate······566
21.3 mod_sofia······567
21.3.1 模块加载······568
21.3.2 Sofia 的加载及通话建立······568
21.4 小结······576

第 22 章 FreeSWITCH 二次开发······577
22.1 给 FreeSWITCH 汇报 Bug 和打补丁······577
22.1.1 修复内存泄露问题······577
22.1.2 给中文模块打补丁······578
22.1.3 给 FreeSWITCH 核心打补丁······579
22.1.4 高手也会犯错误······581
22.1.5 汇报严重的问题······581

22.1.6 给 Sofia-SIP 打补丁 …… 582
22.1.7 给现有 App 增加新功能 …… 584
22.1.8 给 FreeSWITCH 增加一个新的 Interface …… 586
22.2 写一个新的 FreeSWITCH 编解码模块 …… 587
22.3 从头开始写一个模块 …… 589
22.3.1 初始准备工作 …… 589
22.3.2 写一个简单的 Dialplan …… 590
22.3.3 增加一个 App …… 592
22.3.4 写一个 API …… 593
22.3.5 小结 …… 594
22.4 使用 libfreeswitch …… 594
22.4.1 自己写一个软交换机 …… 594
22.4.2 使用 libfreeswitch 提供的库函数 …… 595
22.4.3 其他 …… 599
22.5 调试跟踪 …… 600
22.6 小结 …… 601

附录 A FreeSWITCH 开源软件社区指南 …… 603

附录 B 关于 FreeSWITCH 常用术语翻译的意见 …… 605

附录 C 模块列表 …… 607

附录 D FreeSWITCH 架构详细示意图 …… 614

后记 …… 615

第一部分

基　础　篇

自己动手搭建个人的软交换环境是一件激动人心的事。本篇就带领大家一步一步来完成这件事。当然，在这之前我们首先要一起回顾一下电话交换网络和电话交换技术以及相关的历史，并探讨交换网络的现状和未来；同时，学习一些电信领域的基础知识，熟悉一些基础的电信业务；然后，我们再一起体验 FreeSWITCH 的安装和配置；在对 FreeSWITCH 有了直观的印象之后，我们再详细解释 FreeSWITCH 的设计思想和理念、基本概念和理论、运行和维护的方法，以及重要的协议和模块等。

读完本篇，你可以为成为一名合格的 FreeSWITCH 维护工程师打好坚实的基础。

第 1 章

PSTN 与 VoIP 基础

一说起 VoIP，给人的感觉好像有些抽象，或许大家对“网络电话”这个词更熟悉一些。VoIP 的英文原意是 Voice Over IP，即承载于 IP 网上的语音通信。大家熟悉家庭上网所使用的 ADSL 吧？或许有些人还记得前些年用过的吱吱叫的老“猫”（Modem，调制解调器）。技术日新月异，之前的技术都是用电话线上网，现在 VoIP 技术使我们可以在网上打电话。

所谓“温故而知新”，在学习任何新东西以前，我们最好都了解一下它的历史，以做到心中有数。所以在了解 VoIP 之前，我们先来看一下 PSTN。

PSTN（Public Switched Telephone Network，公共交换电话网）就是我们日常打电话所使用的电话网络。有时，人们喜欢对新鲜的事物刨根问底，却对司空见惯的东西不求甚解。在此，为了便于读者了解本书所述的技术，我们一起来回顾一下交换机的起源和演变过程、探索一下电话网的现状和发展方向，同时一起学习一些与 PSTN 有关的基础知识和专业术语等。

1.1 PSTN 起源与发展

在漫长的通信历史长河中，PSTN 及电话交换技术的发展经历了很多阶段，如从直接控制到间接控制再到公共控制、从人工交换到自动交换、从电子交换到程控交换、从模拟到数字、从电路交换到分组交换、从“硬”交换到软交换等。下面我们分别介绍。

1.1.1 最早的电话网

第一次语音传输是苏格兰人亚历山大·贝尔（Alexander Granham Bell）⊖在 1876 年用振铃电路实现的。而在那之前，普遍认为烽火台是最好的远程通信方式。

用振铃电路实现通话功能的时代是没有电话号码的，相互通话的用户之间必须由物理线

⊖ 1847 年 3 月 3 日 — 1922 年 8 月 2 日。1870 年贝尔移民到加拿大，一年后到美国。1882 年加入美国国籍。参见：https://zh.wikipedia.org/wiki/ 亚历山大·格拉汉姆·贝尔。

路连接；并且，在同一时间只能有一个用户讲话（半双工）。发话方通过话音振动来激励电炭精麦克风从而产生电信号，电信号传到远端后通过振动对方的扬声器发声，从而传到对方的耳朵里。

由于每对通话的个体之间都需要单独的物理线路，如果整个电话网上有 10 个人，而某人想与另外 9 个人通话，他就需要铺设 9 对电话线。同时整个电话网上就需要 10 × (10–1)/ 2=45 对电话线，如图 1-1 所示。

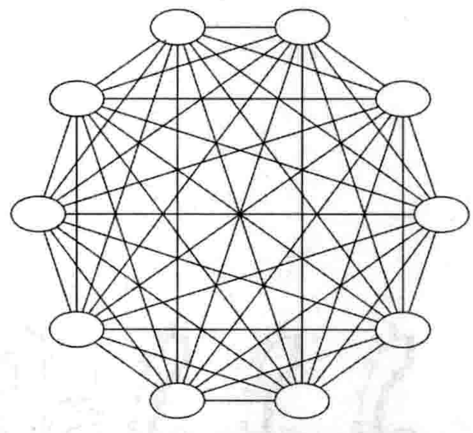


图 1-1 电话网上有 10 个人通话的情况

1.1.2 人工电话交换时代

随着时代的发展，对电话有需求的用户越来越多，甚至普通家庭都希望拥有自己的电话。但是，为每对欲通话的家庭之间都铺设电话线是不可能的。因此，一种称为交换机（Switch，又称 Exchange）的设备诞生了。它位于整个电话网的中心，用于连接每个用户。用户想打电话时，先拿起电话连接到管理交换机的接线员，由接线员负责接通到对方的线路。这便是最早的电话交换网（见图 1-2），交换接续工作全部由人工完成。通过使用交换机，交换网上需要的线路大大减少了。

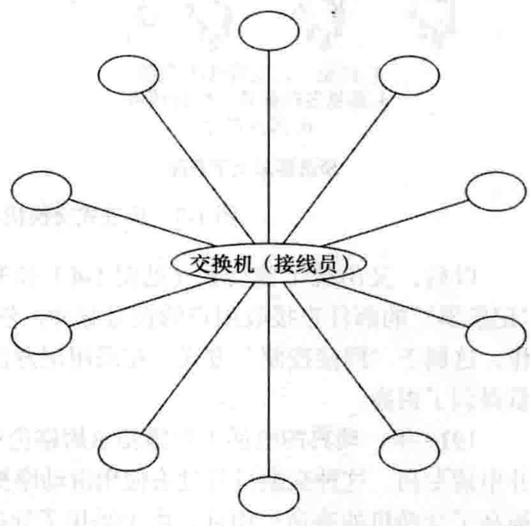


图 1-2 人工电话交换网

1.1.3 自动电话交换时代

1889 年到 1891 年期间，美国阿尔蒙 · B · 史端乔（Almon B Strowger）发明了步进式自动电话交换机。有趣的是，他发明自动交换机的目的并不是为了把接线员从繁忙的人工交换机上解放下来，而是来源于另外一则故事：他本是一个殡仪馆老板。他发觉每当城里发生死亡事件，用户给“殡仪馆”打电话时，不知接线员是有意还是无意的，总会把电话接到另一家殡仪馆。这使史端乔非常郁闷，发誓要将电话交换自动化。功夫不负有心人，史端乔凭他那过人的聪明和毅力，终于发明了一种步进式的自动电话交换机，并申请了专利。人们为了纪念他，故又称这种电话交换机为“史端乔交换机”（见图 1-3）。这种交换机的特点是由用户话机的拨号脉冲直接控制交换机的动作，属于“直接控制”方式。

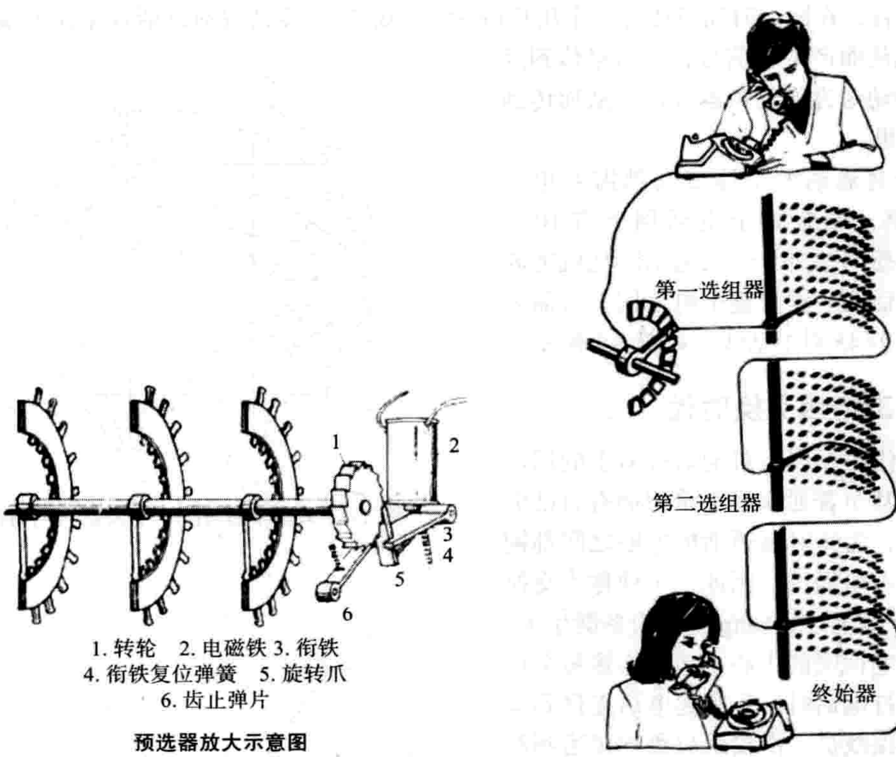
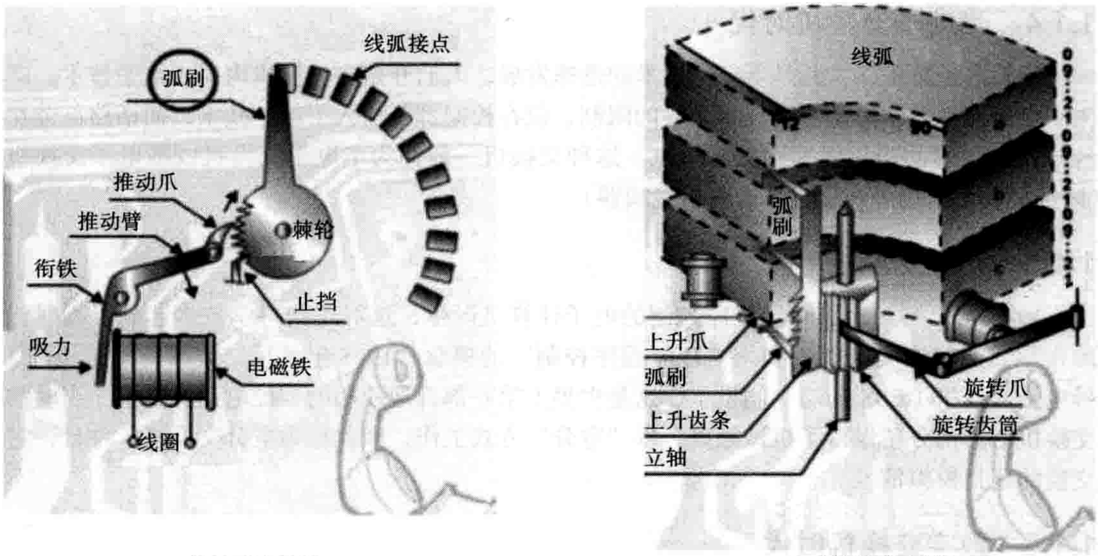


图 1-3 步进式交换机 4 位拨号电话中继示意图

以后，又出现了旋转式（见图 1-4）和升降式的交换机。这类交换机采用了一个称为“记发器”的部件来接收用户的拨号脉冲，然后将其通过译码器译成电码来控制接线器的动作，这属于“间接控制”方式。在采用记发器后，增加了选择的灵活性，从而使交换机的容量得到了提高。

1919 年，瑞典的电话工程师帕尔姆格伦和贝塔兰德发明了“纵横制接线器”(见图 1-5)，并申请专利。这种交换机将过去使用滑动摩擦方式的触点改成了压接触，减少了磨损，从而提高了交换机的寿命；而且，由于采用了导电性好的贵金属（如银）做金属触点，也大大提高了接触的可靠性。这种交换机的另一个特点是把控制部分和话路部分分开。控制部分由标志器和记发器来完成，称为公共控制。公共控制对用户拨号盘要求低，而中继部署的灵活性大大提高。

瑞典和美国分别在 1926 年和 1938 年开通了纵横制交换机，接着，法国、日本和英国也相继生产出纵横制交换机。



a) 旋转型选择器　　b) 上升旋转型选择器

图 1-4　旋转式交换机

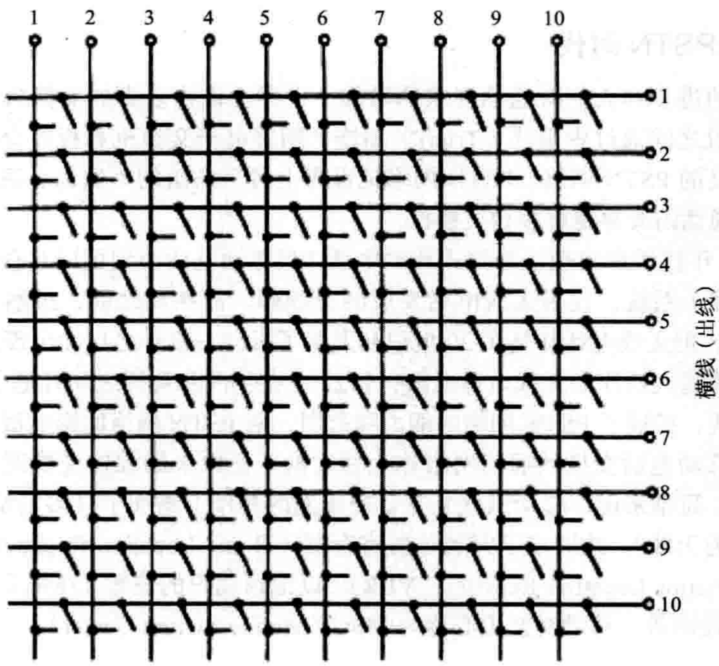


图 1-5　纵横接线器交叉点示意图

1.1.4 半电子交换机时代

随着电子技术，尤其是半导体技术的迅速发展，人们开始在交换机内引入电子技术，产生了电子交换机。由于当时技术条件的限制，仅在控制部分引入了电子技术，而话路部分在较长的一段时间内仍然采用机械触点。这种交换机一般称为半电子交换机或准电子交换机（区别是后者采用了速度较快的笛簧接线器）。

1.1.5 空分交换机时代

1946 年，第一台由存储程序控制的电子计算机诞生，这对现代科学技术起到了划时代的作用。通过在交换机中引入“存储程序控制”的概念，1965 年 5 月，美国贝尔系统的 1 号电子交换机（ESS No.1）问世了，这是世界上第一部开通使用的程控电话交换机。当时的交换机话路部分还保留了机械触点，以“空分”方式工作，因此称为空分交换机。并且，它交换的还是模拟信号。

1.1.6 数字交换机时代

20 世纪 60 年代初以来，脉冲编码调制（PCM）技术成功地应用于传输系统中，它通过将“模拟”的信号数字化，提高了通话质量、增加了传输距离，同时，节约了许多线路成本。1970 年，法国开通了世界上第一部程控数字交换机 E10，开始了数字交换的新时代。

1.1.7 现代 PSTN 时代

随着技术的进步和人们对通信要求的增加，世界上许许多多的交换机间也需要相互通信，这些交换机之间通过中继线（Trunk）相连，随着电子交换机和程控交换机的发展，便出现了现代意义的 PSTN 网络。PSTN 网络把世界上各个角落的人们都联系在了一起，很显然，有时一个通话需要穿越好多台交换机。

20 世纪 70 年代后期出现了蜂窝式移动电话（当移动电话小到可以拿在手里的时候就开始叫“手机”了）系统，这是无线电话发展的里程碑。而在此之前，虽然无线电通信是在 1895 年发明的，但无线电话却是在 20 世纪初发明了真空三极管之后才出现的。1915 年首次成功地实现了跨越大西洋的无线电话通信；1927 年在美国和英国之间开通了商用无线电话。移动电话的出现，扩展了 PSTN 网络的能力和范围，对 PSTN 网络的影响极其深远。

专门用于移动电话交换的通信网络称为移动网，而原来的程控交换网则称为固定电话网，简称固网。简单来说，移动网就是在普通固网的基础上增加了许多基站（Base Station，可以简单理解为天线），并增加了归属位置寄存器（Home Location Register，HLR）和拜访位置寄存器（Visitor Location Register，VLR），以记录用户的位置（在哪个天线的覆盖范围内）、支持异地漫游等。移动交换中心称为 MSC（Mobile Switch Center）。

1.1.8 下一代网络及 VoIP 时代

随着分组交换技术的成熟及因特网的发展，人们认识到了将原始的基于电路交换的语

音网络与基于分组交换的因特网络进行融合（即语音通信和数据通信相结合）的必要性，因此一个称为 NGN（Next Generation Network）的概念被提了出来。NGN 是指下一代网络，ETSI[1]对它的定义是："NGN 是一种规范和部署网络的概念，即通过采用分层、分布和开放业务接口的方式，为业务提供者和运营者提供一种能够逐步演进的策略，实现一个具有快速生成、提供、部署和管理新业务的平台。"[2]

在 NGN 提出之后，经过数年的研究和探索，人们提出了各种 NGN 的解决方案，但最终基本上都统一到了 IMS（IP Multimedia Subsystem）技术[3]。IMS 运行于标准的 IP 网络上，使用一种基于第三方伙伴计划（3GPP）的 SIP 标准的 VoIP 实现方式。IMS 的目标不仅是在现有网络基础上提供新的业务，它还要能提供在未来的因特网上能够承载的所有的业务。

当然，IMS 属于核心交换层的技术，它全部基于 IP 网络，但在接入层，目前的语音大部分还是基于电路接入的方式接入的，因此，这就要求在一定时间内 IMS 在接入层要继续兼容电路接入。同时，在无线移动通信领域，随着移动通信技术的成熟及众多智能终端的出现，对高速 IP 网络的要求也就越来越迫切。最新的 3G[4]、4G[5]技术就是应此要求而产生的。未来的通信中要完全取消低效的电路传输及电路交换，而全部集中到 IP 通信上来，也就催生了一个新的无线通信标准 LTE[6]。LTE 的定义是长期演进而来的，其为现代的手机及其他移动设备提供了高速的数据通信手段，逐步实现了全 IP 交换。

关于通信网络的演进，简单来说，在无线方面体现为从 GSM/CDMA/UMTS 等向 LTE 发展，在核心网方面则体现为从电路交换向 IMS 发展。过去几年，围绕 LTE 语音曾经出现过多种观点、技术和演进路线。由于 LTE 标准不再支持用于支撑 GSM、UMTS 和 CDMA2000 网络下语音传输的电路交换技术，它只能进行全 IP 网络下的分组交换，因此随

---

[1] European Telecommunications Standards Institute，即欧洲电信标准化协会，官方网站是 http://www.etsi.org。

[2] 原文是：NGN is a concept for defining and deploying networks, which, due to their formal separation into different layers and planes and use of open interfaces, offers service providers and operators a platform which can evolve in a step-by-step mannerto create, deploy and manage innovative services. 不过，这个概念的定义跟它的名字一样令人迷惑。事实上，这个概念最早大约是在 1997 年提出来的，所以，"下一代"中的"下"早就过时了。

[3] 一般来说，NGN 就应该称为软交换，而有的人也说 IMS 是更软的软交换。读者到后面就能了解到，与"更软的软交换"相比，FreeSWITCH 更是用纯软件实现的，或许可以称做是"更更软的软交换"了。当然软交换的"软"并不是取决于它是否用"软件"来实现，而是更强调基于分组网的交换网络中的呼叫控制功能与媒体网关的分离。

[4] 3G 是指第三代移动通信技术，与第一代（1G、模拟系统）和第二代（2G、以 GSM 为代表的数字系统）移动通信技术相比，它主要是支持高速 IP 数据传输。详见 http://zh.wikipedia.org/wiki/3G 。

[5] 4G 是指第四代移动通信技术，是 3G 之后的延伸，从技术标准的角度看，按照 ITU 的定义，静态传输速率达到 1Gbps，用户在高速移动状态下可以达到 100Mbps，就可以称为 4G 的技术。详见 http://zh.wikipedia.org/wiki/4G。

[6] 参见 http://zh.wikipedia.org/wiki/ 长期演进技术。

着 LTE 网络的部署，运营商需选择 VoLTE[㊀]、CSFB[㊁]、SVLTE[㊂]、OTT[㊃]等方法之一解决 LTE 网络中的语音传输问题。从目前来看，移动网络庞大且复杂，在网络建设初期大部分运营商都选择使用 CSFB 方式建设网络，这种方式便于快速部署系统。当然，CSFB 只是过渡时期的临时解决方案。从长远来看，VoLTE 及其他几种方案更符合未来网络的发展方向。据报道，中国移动已于 2013 年 6 月发布了 VoLTE 技术白皮书，并计划于 2014 年下半年展开大规模商用。未来通信网络将走向何方，让我们拭目以待。

1.2 电话实现技术

电话系统的发展与科技的进步是分不开的。在本节，我们来介绍一些关键的电话技术及专业术语。

1.2.1 电话号码

我们的生活已经离不开电话，而要打电话就离不开电话号码。但好多人对天天使用的电话号码既熟悉又陌生，因此，在这里我们也补充一些现行的电话网中电话号码的知识。值得注意的是，这些号码的出现大部分与业务内容和行政区划有关，也有一些历史背景，但限于篇幅，我们就不深入讨论了。

1. 固定电话号码

现行的电话网中采用 E.164[㊄]号码格式。以我国固话电话的编号规则为例，我们先来看本地号码。

一般来说，在比较大的省市（如北京、上海等）使用 8 位号码，比较小的省市使用 7 位号码。以北京的电话号码为例，它由 8 位数字组成，表示为 ABCD EFGH，其中，ABCDE

㊀ VoLTE（Voice Over LTE，LTE 网络直传）：该方案基于 IP 多媒体子系统（IMS）网络，配合 GSMA 在 PRD IR.92 中制定的 LTE 控制和媒体层面的语音服务标准。使用该方案意味着语音将以数据流形式在 LTE 网络中传输，所以无须调用传统电路交换网络，旧网络无须保留。简单来说，VoLTE 就是 LTE 网络环境下的 VoIP 业务。

㊁ CSFB（Circuit Switched Fallback，电路交换网络支援）：该方案中的 LTE 网络将只用于数据传输，当有语音拨叫或呼入时，终端将使用原有电路交换网络。该方案只需运营商升级现有 MSC 核心网而无须建立 IMS 网，因此运营商可以较迅速地向市场推出网络服务。也由于语音通话需要切换网络才能使用的缘故，通话接续时间将被延长。

㊂ SVLTE（Simultaneous Voice and LTE，LTE 与语音网同步支持）：该方案使用可以同时支持 LTE 网络和电路交换网络的终端，使得运营商无须对当前网络作太多修改。但这同时意味着终端价格的昂贵和电力消耗的迅速。

㊃ OTT（Over-The-Top）：一种将服务建于网络之上的方式，如用 Skype 来提供 LTE 的语音。LTE 具备高带宽、低时延、永远在线、全 IP 等特点，为 OTT 的发展带来了天然的便利，使得 OTT 语音几乎没有壁垒。但应当看到，现在乃至将来很长时间里，语音业务都是移动运营商的最主要的收入来源，把 LTE 的语音业务完全交给 OTT 是一种非常激进的观点，在电信领域得到的支持并不多。

㊄ 参见 http://en.wikipedia.org/wiki/E.164。

称为一个千群，即该群可以包含 1000 个电话号码，同理 ABCD 称为一个万群。小规模的交换机可能只包含几个千群，而大规模的可能包含几个万群。

如果在不同的省市之间打电话，则称为长途电话，一般需要通过长途局进行路由。打长途电话需要在本地号码前加上长途区号。根据城市的大小不同，我国电话区号的长度一般是 2 位到 3 位，如：

```
10  北京
20  广州
21  上海
755 深圳
535 烟台
```

读者可能会问，北京的长途区号不是 010 吗？答案是——“不是！”许多人会错误地认为北京的区号是 010，这也正是作者要写这一节的原因。

由于各地区号的长度不同，而且数字也不规则，因此，为了区分长途电话和本地电话，我国规定，在拨打长途电话前除了要加上区号外，还要在最前面加“0”。因此，“0”是国内长途字冠，它本身不是区号的一部分。为了帮助读者理解，我们先考虑国际长途的情况。为了区分国内长途与国际长途，我国规定，国际长途要在国家代码前加两个 0。因此，如果拨打美国的号码，如：1-234-567-890，则需要拨 00-1-234-567-890。其中，1 为美国的国家代码，我们稍后还会讲到。

交换机的拨号规则遵循短号优先的原则，拨号时不加 0 就认为是本地号码，加一个 0 就认为是国内长途，两个 0 是国际长途。我国的国家代码是 86，因此，一个北京号码的国际表示是：86 10 ABCD EFGH。

与我国不同，美国的国际长途字冠是 011，因此如果从美国拨打中国的号码，则需要拨：011 86 10 ABCD EFGH。

类似的，其他国家和地区也有不同的拨号方式，这增加了电话号码书写的复杂性。因此，在实际书写电话号码时，有时会省略国际长途字冠，而统一用“+”号代替，如：+86 10 ABCD EFGH ⊖。

读到这里，读者就可以理解为什么北京的区号不是 010 了吧？另外，我们还知道，使用固定电话拨打异地手机时前边要加拨 0，同样说明 0 并不是区号或手机号码的一部分。

2. 移动电话号码和专用号段

移动电话号码俗称手机号，由于其可移动与漫游的特性，因此与普通固定电话号码略有不同。众所周知，我国的移动电话号码都是以 1 开头的，按不同的运营商来划分。如中国移动的号码由 135、136、137、138、139 等开头；中国联通的由 130、186 等开头，中国电信的由 133、189 等开头。它们后面都是 8 位的号码，因此整个手机号共 11 位，前 3 位称为移动电信运营商专用的号段。

在 2013 年工信部发放虚拟运营商牌照以后，核发 170 号段作为虚拟运营商的专用号段。

⊖ 当然，为了方便阅读，有时还会在电话电码中增加括号和短横线。

3. 短号码

有一类特殊的号码称为短号码，如大家熟知的 110、119、120 等，这些属于公益性的号码（又称紧急号码，供紧急情况下使用），拨打都是免费的。

还有一部分是预付费的电话卡类业务，如 200、201、300 等。拨打这类号码本身不收费，而从关联的电话卡上扣费。

另一些短号码，如 114 等，由于运营商的分拆、合并，有的演变成了 116114；还有一些 5 位的短号码专门供一些大的集团（如银行等）使用，如 95555、95588 等，这些号码不论何时何地拨打，一般只收市话费。

这些短号码的资源比较紧张，一般不会给个人使用。另外还有其他一些短号码，我们就不多讲了。

4. 800 和 400 号码

800 开头的号码是被叫付费的业务，主叫呼叫这些号码是免费的。这些号码主要由一些大的企业集团使用。这类号码都是虚拟的电话号码，在实际呼叫过程中通过查询数据库转换成真正的目的号码。

但是 800 号码有一个致命的缺点，就是用手机打不通，这主要是电信业务的历史原因（主要原因是不同运营商的网间结算）造成的。随着移动业务的发展，手机用户越来越多，因此，出现了 400 业务。这类业务的特点是主被叫分摊付费，主叫付本地通话费，被叫付长途电话费（如果主被叫不在一个城市）。400 业务是手机用户可以呼叫的。

当然，随着时代的发展，以及各运营商提供更加灵活的话务套餐（如主叫免收长途费等），主叫用户对拨打这类号码时对实际资费已不太敏感。拥有 800 和 400 之类的电话号码越来越多地成为了企业实力和社会形象的标志。

5. 北美电话号码分类计划

为了与国内号码进行对比，我们来简单说一下北美电话号码分类计划（North American Numbering Plan ⊖）。

加拿大和美国使用北美电话号码分类计划，其区号由 3 位数字组成，本地号码为 7 位数字，1 为长途接入码，即长途字冠（在有的情况下可以省略长途字冠）。比如，一个完整的号码为 1(ABC) DEF-GHIJ，如果是在本地拨打，则可以直接拨“DEF GHIJ”，如果是拨打长途，则需要先拨长途字冠 1 及区号，即：1(ABC) DEF-GHIJ。

值得一提的是，其中的区号 ABC 如果是 555，除 555-1212 是查号台外，其他的号码都是不存在的。这类号码一般用于电影或戏剧中，防止与真实环境中的电话号码相冲突。

6. 电话号码的书写格式

电话号码就是一长串号码，但有时候为了便于阅读，在写的时候常用连字符“-”、括号、空格等将数字分开，如上一节我们看到美国电话号码的格式。

⊖ 参见 http://zh.wikipedia.org/wiki/ 北美电话号码分类计划。

国内电话号码的书写一般采用如下的方式：

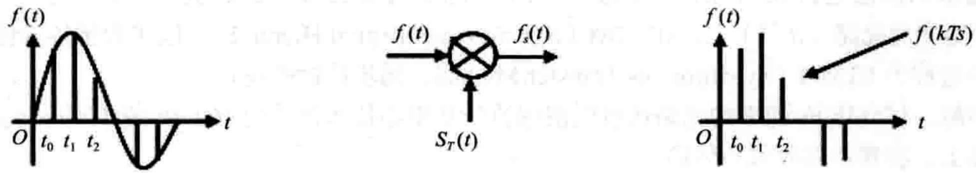
```
(010) ABCD EFGH   （没有国家代码，虽然 0 不是区号的一部分，但是，习惯了）
+86 (10) ABCD EFGH（固话，8 位，国际号码格式）
+86 (535) ABC DEFG（固话，7 位，国际号码格式）
+86 139 ABCD EFGH （手机，国际号码格式）
```

当然，具体的写法没有统一的规定，只要让看到号码的人知道怎么拨打就行了。至于要将电话号码印到名片上，又涉及企业形象设计的问题了，那就另当别论了。

1.2.2 模拟信号与数字信号

模拟（Analog）量是连续的变化的量，如温度、声音等。早期的电话网也是基于模拟交换的。对于人类交流来讲模拟信号是非常理想，但它很容易引入噪声。如果通话双方距离很远，信号会衰减，因而需要对信号进行放大。问题是，信号中经常混入线路的噪声，放大信号的同时也放大了噪声，导致信噪比（信号量与噪声的比例）下降，严重时甚至会难以分辨。

数字（Digital）信号是不连续的（离散的），是按一定的时间间隔（单位时间内抽样的次数称为频率）对模拟信号进行抽样（见图 1-6）得出的一些离散值。然后通过量化和编码过程就可将这些离散值变成数字信号。根据抽样定理[⊖]，当抽样频率是模拟信号最高频率的 2 倍时，就能够完全还原原来的模拟信号。



连续模拟信号　　　　乘法器　　　　离散模拟信号

图 1-6　抽样

1.2.3 PCM

PCM（Pulse Code Modulation）的全称是脉冲编码调制。它是一种通用的将模拟信号转换成以 0 和 1 表示的数字信号的方法。

一般来说，人的声音频率范围在 300~ 3400Hz 之间，通过滤波器将超过 4000Hz 的频率过滤出去，便得到 4000Hz 内的模拟信号。然后根据抽样定理，使用 8000Hz 进行抽样，便得到离散的数字信号。使用 PCM 方法得到的数字信号就称为 PCM 信号，一般一次抽样会得到 16bit 的信息。

为了更有效地在线路上进行传输，通常对 PCM 信号进行一定的压缩。通过使用压缩算

⊖ 又称采样定理或奈奎斯特 · 香农定理，见 http://zh.wikipedia.org/wiki/ 采样定理。

法[⊖]，可以将每一个抽样值压缩到 8bit。这样 1 秒的抽样就得到 8bit × 8000=64000bit 的信号（简称 64kbit），抽样速率（即传输速率）为 64kbit/s [⊜]。

PCM 通常有两种压缩方式：A 律和 μ 律。其中北美使用 μ 律，我国和欧洲使用 A 律。这两种压缩方法很相似，都采用 8bit 的编码获得 12bit 到 13bit 的语音质量[⊜]。但在低信噪比的情况下，μ 律比 A 律略好。A 律也用于国际通信，因此，凡是涉及 A 律和 μ 律转换的情况，都由使用 μ 律的国家负责。

1.2.4 局间中继与电路复用技术

连接交换机（局）的 E1 或 T1 电路称为局间中继。交换机间的消息以及通话数据都是在局间中继上传送的，传统的交换机使用时分复用（TDM）技术将多路通话合并到一条数字中继线上，可以大大节省局间中继线的数量。

利用时分复用技术可以将 32 个 64kbit/s 的信道合并到一条 2Mbit/s（64kbit/s × 32 = 2.048Mbit/s，通俗来说就直接叫一个两兆）的电路上，这样的电路称为一个 E1（在北美和日本，是 24 个 64kbit/s 复用，称为 T1，速率是 1.544Mbit/s）。在 E1 中，每一个信道称为一个时隙。其中，除 0 时隙固定传同步时钟外，其他 31 个时隙最多可以同时支持 31 路电话（有时候会使用第 16 时隙传送信令，这时最多支持 30 路电话）。

随着话务量的增加，交换机之间的电路越开越多，因而需要更高级别的电路复用技术。目前通常的做法是将 63 个 E1 合并到一个 155Mbit/s（$2 \times 63+P=155$，其中 P 是电路复用的开销）速率的光路（光纤）上，在 SDH（Synchronous Digital Hierarchy，同步数字传输体系）技术中这称为 STM-1（Synchronous Transfer Module，同步传输模块）。

当然，155Mbit/s 速率的光路还可以使用波分复用等技术合并到 1Gbit/s 或 10Gbit/s 速率的光路上，实现话路收敛和传输。

1.3 我国电话网结构

我国的电话网由本地网与长途网组成，并通过国际交换中心进入国际电话网。

原有电话网的结构采用了等级制，共有五级，分别用 C1 ~ C5 表示。其中 C1 ~ C4 构成长途电话网，C5（端局）及汇接局（Tm）构成本地网。但随着社会和经济的发展，电话业务量迅速增加，横向话务流量日趋增多，新业务的需求不断涌现，五级网络结构由于转接段数多造成了接续时延长、传输损耗大、网络管理工作复杂、不利于新业务的开展等问题。因此，我国电话网正在由五级电话网向三级电话网转变，即将原来的 C1 级和 C2 级长途交换

⊖ 实际为压扩法，因为有的部分是压缩的，有的是扩张的。目的是给小信号更多的比特位数以提高语音质量。

⊜ bit/s 即比特 / 秒，有时也写作 bps（Bit Per Second）。注意，通常，对于二进制数来说 K 为大写，1K=1024，但此处的 k 为小写，1k=1000。

⊜ 目前我国使用的是 A 律 13 折线特性。基本原理是：使用非均匀量化方法和压缩扩张技术得到 13 段折线。感兴趣的读者可在 Google 上搜索“13 段折线”进行更深入的了解。

中心演变为 DC1，原来的 C3 级和 C4 级交换中心演变为 DC2，以便减少转接段数。下面，我们主要以本地网为例简单讲解一下电话网的结构。

图 1-7 所示的主体部分为我国一地级市固定电话网（称为本地网）的结构。通常，用户话机（如 A、B、C、D）通过一对电话线连接到距离最近的交换机上，该交换机称为端局交换机（一般以区或县为单位）。端局交换机通过局间中继线连接到汇接局[1]。长途电话需要通过所在长途局与其他长途局相连。但根据话务量要求，汇接局也可以直接与其他长途局开通高速直达中继（未画出）。为节省用户线，在一些人口比较集中的地方（如学校、小区），会在端局下再设模块局或接入网，用户话机（如 a、b、c、d、e）则就近接入到模块局（或接入网）上。

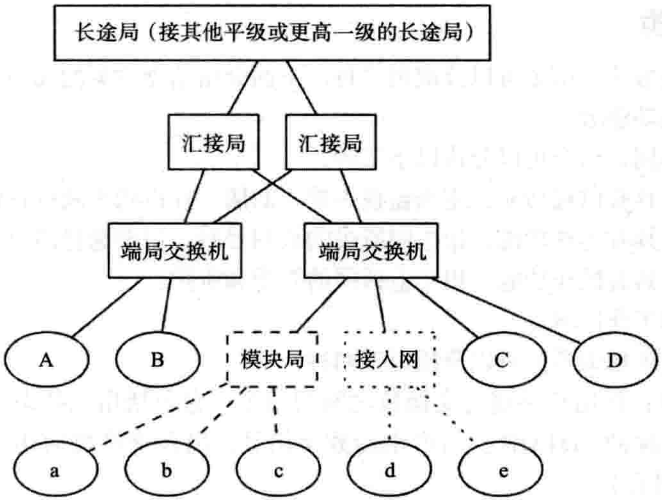


图 1-7 我国固定电话网结构

在我国，智能网一般用于实现电话卡、预付费或 400/800 类业务，而前几年新部署的 NGN（Next Generation Network，下一代网络，一般指软交换）则支持更灵活、更复杂的业务，包括 IM/SMS/MMS 等消息类业务、点对点交互式多媒体 / 文件共享 / 游戏等多方协作交互业务、内容分发业务、广播 / 多播业务、企业托管业务，以及数据检索、数据通信、在线应用、传感器网络、远程控制等各类多媒体业务和数据业务。

但是技术发展很快，为响应国家"光进铜退"[2]的号召，现在已经实现了直接光纤到户，也就是说很快全网都能 IP 化了。新技术代替旧技术，为用户带来了五彩斑斓的业务，但旧技术的一些优点则永远让人怀念，最典型的就是原来的电话线是带电的，家里停电不影响打电话。但光纤是不带电的，停电以后用户就不能打电话了（除非加 UPS）。

---

[1] 为了保证安全，汇接局通常会成对出现，平常实行负荷分担，一台汇接局出现故障，与之配对的汇接局会承担其负责的所有话务。

[2] 用户家里的电话线使用铜双绞线。

在我国的移动网络中，大量部署了 IMS。IMS 具有组网灵活的优点，但涉及的网元和基本概念很多，对于非专业人士来说理解起来有些困难。我们将在本章的最后一节对 IMS 进行专门的讨论。

1.4 信令

用户设备（如话机）与端局交换机之间，以及交换机与交换机之间需要进行通信。这些通信所包含的信息有（但不限于）用户、中继线状态、主叫号码、被叫号码、中继路由的选择等。我们把这些消息称为信令（Signaling）。

1.4.1 信令分类

按照不同分类方式，信令可以分成很多种。下面介绍信令主要的几种分类方式。

（1）按信令的功能分

按照功能的不同，信令可以分成以下三种：

- 线路信令：具有监视功能，用来监视主被叫的摘、挂机状态及设备忙闲。
- 路由信令：具有选择功能，指主叫所拨的被叫号码，用来选择路由。
- 管理信令：具有操作功能，用于电话网的管理和维护。

（2）按信令的工作区域分

信令按照工作区域的不同可以分成以下两种：

- 用户线信令：是用户终端与交换机之间的信令。它包括用户状态（摘、挂机）信号、用户拨号（脉冲、DTMF ）所产生的数字信号，以及交换机向用户终端发送的信号（铃流、信号音）。
- 局间信令：是交换机和交换机之间的信令，在局间中继线上传送，用来控制呼叫接续和拆线。

注意

用户线信令少而简单，局间信令多而复杂。

（3）按信令的信道分

按照信道的不同，信令可以分为以下两种：

- 随路信令：信令和话音在同一条话路中传送。
- 公共信道信令：以时分方式在一条高速数据链路上传送一群话路。

随路信令传送速度慢，信息容量有限（传递与呼叫无关的信令能力有限）；公共信道信令传送速度快、容量大，具有改变或增加信令的灵活性，便于开放新业务。

（4）其他分类

另外，信令还可分为带内信令和带外信令、模拟信令和数字信令、前向信令和后向信令、线路信令和记发器信令等，我们在这里就不多解释了，有兴趣的读者可以自行搜索相关

的关键词进一步学习。

下面我们分别对一些重要的信令进行简单介绍。

1.4.2 用户线信令

从用户终端（通常是话机）到端局交换机之间经常需要传送一些控制信息，如用户摘机、挂机、拨号、主叫号码显示等，这些信息称为用户线信令。用户线信令可以通过模拟或数字信号传递。

对于普通的话机，线路上传送的是模拟信号，因此信令只能在电话线路上传送，这种信令称为带内信令。话机通过电压变化来传递摘、挂机信号；通过 DTMF（Dual Tone Multi Frequency，双音多频㊀）传送要拨叫的电话号码。另外，也可以通过移频键控（Frequency Shift-keying，FSK）技术来支持主叫号码显示，俗称来电显示（Caller Line Identification Presentation，Caller ID 或 CLIP，主叫线路识别提示）。

与普通电话不同，ISDN（Integrated Service Digital Network，综合业务数字网）在用户线上传送的是数字信号。它的基本速率接口（Base Rate Interface，BRI）使用 144kbit/s 的 2B+D 信道——两个 64kbit/s 的 B 信道及一个 16kbit/s 的 D 信道。其中 B 信道一般用来传输话音、数据和图像，D 信道用来传输信令或分组信息。

2B+D 的 ISDN 最初是为了解决用户线上的语音与数据同步传输问题。但事实上，2B+D 的 ISDN 并不像传说中那么美，而且需要专门的 NT1 终端设备，在我国并没有发挥出它应有的作用，后来很快被 ADSL 技术取代了。

1.4.3 局间信令

交换机与交换机之间也需要传送控制信号，用于话路的建立、释放等，这些控制信号就称为局间信令。局间信令主要在局间中继上传送，传送局间信令的电路称为信令链路。一般来说，一条信令链路通常只占用一个 64kbit/s 的时隙。一条信令消息通常只有几十或上百个字节，一条 64kbit/s 的电路足以容纳成千上万路电话所需要的信令。但随着技术的进步、话务量的上涨以及更多增值业务的出现，完成一次通话需要更多的信令消息，因此出现了 2Mbit/s 速率的信令链路，即整个 E1 链路上全部传送信令。

目前在传统的 PSTN 网络中常见的局间信令有 ISDN PRI（Primary Rate Interface，基群速率接口）信令和七号信令。PRI 信令和话路在同一个 E1 上传送，通常使用第 16 时隙，而 0 时隙传送同步信号，其他 30 个时隙可传输通话信息，因此又称 30B+D。与 PRI 信令不同，七号信令除可以与话路在同一个 E1 上传送外，还可以在专门的用于传送信令链路的 E1 中继上传送，因而它组网更加灵活，支持更大的话务量。我们将在下一节专门讲解七号信令。

支持七号信令的每个通信设备都需要有一个全局唯一信令点编码，而信令点编码资源是

㊀ 话机上每个数字或字母都可以发送一个低频和一个高频信号相结合的正弦波，交换机经过解码即可知道对应的话机按键。

比较有限的。因而，七号信令主要在运营商的设备上使用，而运行商与用户设备（如 PBX）一般使用 PRI 信令对接。

1.4.4 七号信令

七号信令（Signaling System No. 7，SS7）是我国目前使用的主要信令方式，用于局间通信。我国的电话网络中有专门的七号信令网。

在此，我们先来看一次简单的固定电话的通话流程。如图 1-8 所示，用户 a 摘机，与其相连的交换机 A 根据电压、电流的变化检测到 a 摘机后，即向 a 发送拨号音，同时启动收号程序。a 听到拨号音后开始拨号，待交换机 A 收齐号码后，即查找路由，发送 IAM（Initial Address Message，初始地址消息）给交换机 B。B 向 A 发 ACM（Adress Complete Message，地址全消息）并通知用户 b（b 的话机）振铃，A 向 a 送回铃音。这时如果 b 接听电话，则 B 向 A 发送 ANC（Answer Charge，应答计费消息），a 与 b 开始通话，同时 A 对 a 进行计费⊖。

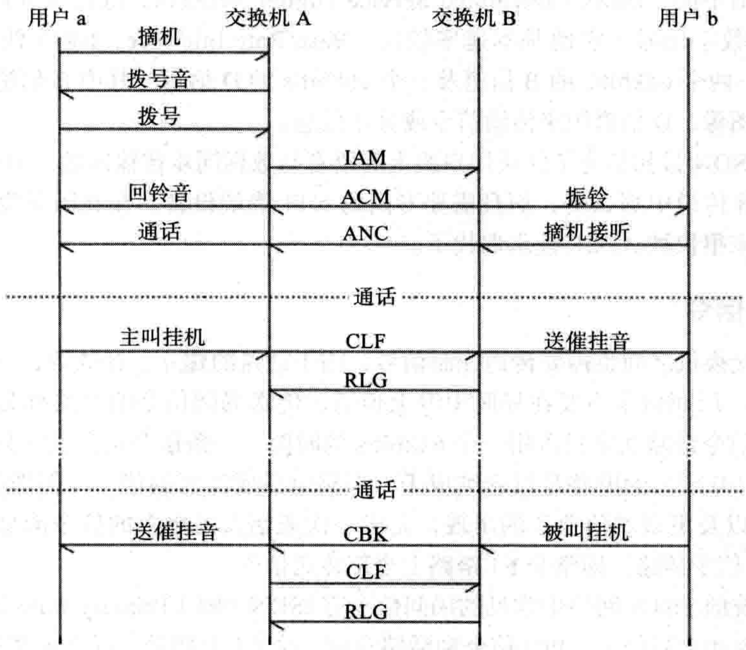


图 1-8 七号信令局间呼叫流程

通话完毕，如果主叫挂机，则本端交换机 A 向对端 B 发送 CLF（Clear Forward，前向释放消息），B 向 A 返回 RLG（ReleaseGard，释放监护消息），并向 b 传送催挂音（嘟嘟嘟……）。

如果被叫挂机，则 B 向 A 发送 CBK（Clear Backword，后向释放消息），A 回送 CLF，最后 B 返回 RLG。

⊖ B 也可以对 A 计费，这种收费方式称为中继计费，主要用于局间结算（话费）。如果 A 和 B 分属于不同的运营商（如移动和电信），则称为网间结算。

上面在交换机 A 与 B 之间传递的为七号信令中的 TUP（Telephone User Part，电话用户部分）。目前，由于 ISUP（ISDN User Part，ISDN 用户部分）能与 ISDN 互联并提供比 TUP 更多的能力和服务，故其已基本取代 TUP 成为我国七号信令网采用的主要信令方式。ISUP 信令与 TUP 互通时的对应关系如图 1-9 所示。

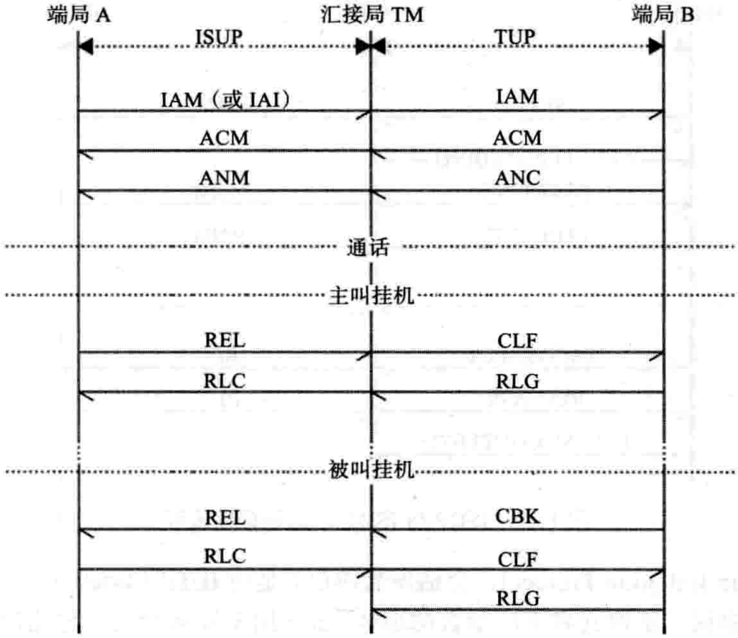


图 1-9　ISUP 与 TUP 互通信令流程

图 1-9 所示为端局 A 与端局 B 经过汇接局 TM 汇接通信中 ISUP 与 TUP 信令的例子。ISUP 信令的初始地址消息有 IAM 和 IAI（IAM With Additional Information，带附加信息的 IAM）两种，后者能提供更多的信息（如主叫号码等）。另外 ISUP 信令的拆线信号不分前后向，只有 REL（Release，释放）和 RLC（Release Complete，释放完成）。

在上一节中我们提到了 ISDN PRI 信令，出于完整性的考虑，这里我们也来看一下 ISUP 与 ISDN PRI 信令互通的例子。ISDN PRI 使用 SETUP/CONNECT/RELEASE 消息分别对应 ISUP 中的 IAM/ANM/REL 消息，如图 1-10 所示。

1.4.5　H.323 与 SIP 信令

H.323 与 SIP 属于 VoIP 领域的通信信令，它们适用于用户线信令和局间信令，由于 IP 终端比普通话机更加智能，因此这些信令在用户线信令及局间信令使用方式上已没有太大区别。

H.323 是 ITU 多媒体通信系列标准 H.32x 的一部分，该系列标准使得在现有通信网络上进行视频会议成为可能，其中，H.320 是在 N-ISDN 上进行多媒体通信的标准；H.321 是

在 B-ISDN 上进行多媒体通信的标准；H.322 是在有服务质量保证的 LAN 上进行多媒体通信的标准；H.324 是在 GSTN 和无线网络上进行多媒体通信的标准。H.323 为现有的分组网络 PBN（如 IP 网络）提供多媒体通信标准，若和其他的 IP 技术（如 IETF 的资源预留协议 RSVP）相结合，就可以实现 IP 网络的多媒体通信。

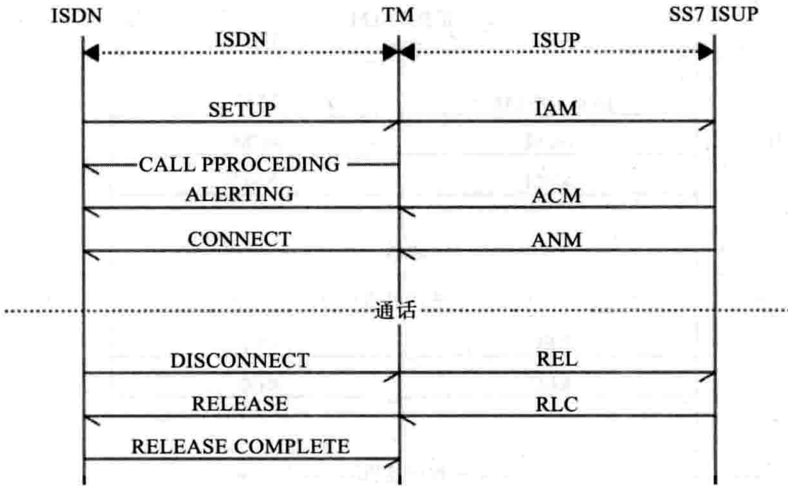


图 1-10 ISUP 与 ISDN 互通的信令流程

SIP（Session Initiation Protocol，会话发起协议）是由 IETF（Interne 工程任务组）提出的 IP 电话信令协议。正像其名字所隐含的那样，SIP 用于发起会话，它能控制多个参与者参加的多媒体会话的建立和终结，并能动态调整和修改会话属性，如会话带宽要求、传输的媒体类型（语音、视频和数据等）、媒体的编解码格式、对组播和单播的支持等。

H.323 和 SIP 设计之初都是作为多媒体通信的应用层控制（信令）协议，目前一般用于 IP 电话。它们能实现的信令功能基本相同，也都利用 RTP 作为媒体传输的协议。但两者的设计风格截然不同，这是由于推出它们的两大阵营（电信领域与 Internet 领域）都想沿袭自己的传统。H.323 是由国际电信联盟提出来的，它企图把 IP 电话当作众所周知的传统电话，只是传输方式由电路交换变成了分组交换，就如同模拟传输变成数字传输、同轴电缆传输变成了光纤传输。而 SIP 侧重于将 IP 电话作为 Internet 上的一个应用，较其他应用（如 FTP，E-mail 等）增加了信令和 QoS 的要求。 H.323 推出较早，协议发展得比较成熟，由于其采用的是传统的实现电话信令的模式，故便于与现有的电话网互通，但相对复杂。SIP 借鉴了其他 Internet 标准和协议的设计思想，有其突出的优点。

- ❑ SIP 是基于文本的协议，而 H.323 采用基于 ASN.1 和压缩编码规则的二进制方法表示其消息，因此，SIP 对以文本形式表示的消息的词法和语法分析比较简单。
- ❑ SIP 会话请求过程和媒体协商过程等是一起进行的，因此呼叫建立时间短，而在 H.323 中呼叫建立过程和进行媒体参数等协商的信令控制过程是分开进行的。

- H.323 为实现补充业务定义了专门的协议，如 H.450.1、H.450.2 和 H.450.3 等，而 SIP 只要充分利用已定义的头域，必要时对头域进行简单扩展就能很方便地支持补充业务或智能业务。
- H.323 进行集中、层次式控制。尽管集中控制便于管理（如便于计费和带宽管理等），但是当用于控制大型会议电话时，H.323 中执行会议控制功能的多点控制单元很可能成为瓶颈。而 SIP 类似于其他的 Internet 协议，设计上为分布式的呼叫模型服务，具有分布式的组播功能。

关于 SIP 信令我们将在第 8 章详细讲解，读者可以先在这里参考一下图 1-11 所示的 ISUP 与 SIP 间的信令转换关系图，以便建立一个直观的印象。

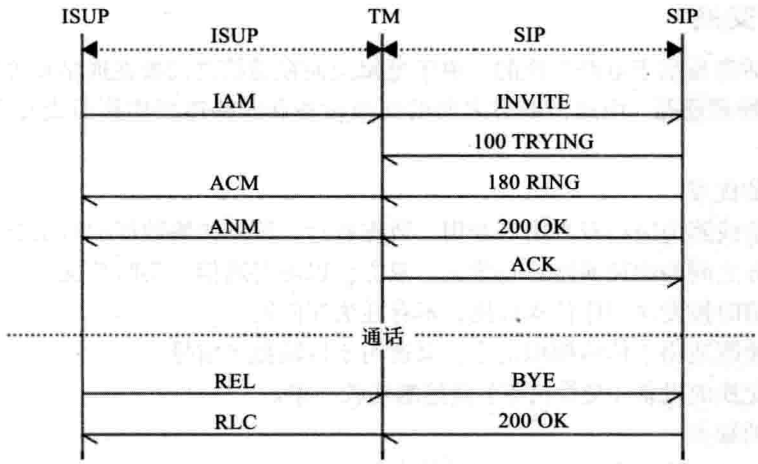


图 1-11　ISUP 与 SIP 互通的信令流程

1.5　媒体

信令主要传输一些控制信号，而通信双方需要听到的是对方的语音数据，这些语音数据就称为媒体（Media）。随着通信系统能力的提高及更加高级、智能的终端设备的出现，通信系统所能传输的媒体类型也越来越丰富，比较典型的有我们刚才提到的语音，还有视频、文字信息（短消息）、传真等。

在 SIP 通信中，除文字外，媒体都是在 RTP 协议中传输的。由于媒体一般都是持续传输的，因此又称 RTP 流。

举一个通俗一点例子：假设张三乘火车从北京到南京，那么张三就是媒体。而火车称为承载（或载体，Carrier）。为了保证正常运行，火车需要听从控制信号的指挥，而这些控制信号就相当于信令。如果信令出了问题，媒体就可能无法正常到达了。当然这个比方可能不十分恰当，你也可以认为张三和火车都是媒体，只不过张三被火车“包”了一层，而铁路才

是真正的载体。不管怎么说，在 SIP 通信中，SIP 相当于这里的火车控制信号，RTP 流中的语音数据相当于很多张三，他们被 RTP 又“包”了一层，通过以太网承载到达目的地。

1.6 电路交换与分组交换

在传统的电路交换中，两个通信节点间需要建立一个专用通路，这会导致电路利用率较低。而报文交换以报文作为数据交换的单位，携带目标地址、源地址等信息，在节点间采用存储转发的方式，不需要建立专门的通信线路，可以大大提高通信线路的利用率。分组交换是报文交换的特殊情形，下面我们来介绍一下电路交换和分组交换⊖。

1.6.1 电路交换

传统的电话都是基于电路交换的。由于电路交换在通信之前要在通信双方之间建立一条被双方独占的物理通路（由通信双方之间的交换设备和链路逐段连接而成），因而有以下优缺点。

电路交换的优点：

- 由于通信线路为通信双方用户专用，数据直达，所以传输数据的时延非常小。
- 通信双方之间的物理通路一旦建立，双方可以随时通信，实时性强。
- 双方通信时按发送顺序传送数据，不存在失序问题。
- 电路交换既适用于传输模拟信号，又适用于传输数字信号。
- 进电路交换的设备（交换机等）及控制均较简单。

电路交换的缺点：

- 电路交换的平均连接建立时间对计算机通信来说较长。
- 建立电路交换连接后，物理通路被通信双方独占，即使通信线路空闲，也不能供其他用户使用，因而信道利用率低。
- 在进行电路交换时，数据直达，不同类型、不同规格、不同速率的终端很难相互进行通信，也难以在通信过程中进行差错控制。

1.6.2 分组交换

我们熟悉的 IP 交换采用的就是分组交换的方式。它仍采用存储转发的传输方式，但将一个长报文先分割为若干个较短的分组，然后把这些分组（携带源、目的地址和编号信息）逐个地发送出去，因此分组交换除了具有报文交换的优点外，与报文交换相比其还有以下优缺点。

分组交换的优点：

- 加快了数据在网络中的传输速度。因为分组是逐个传输，可以使后一个分组的存储操作与前一个分组的转发操作并行，这种流水线式传输方式减少了报文的传输时间。此

⊖ 参考自：http://baike.baidu.com/view/188266.htm。

外，传输一个分组所需的缓冲区比传输一份报文所需的缓冲区小得多，这样因缓冲区不足而等待发送的几率及等待的时间也必然少得多。

- 简化了存储管理。因为分组的长度固定，故相应的缓冲区的大小也固定，在交换节点中存储器的管理通常被简化为对缓冲区的管理，相对比较容易。
- 减少了出错几率和重发数据量。因为分组较短，其出错几率必然减少，每次重发的数据量也就大大减少，这样不仅提高了可靠性，也减少了传输时延。
- 由于分组短小，更适用于采用优先级策略，便于及时传送一些紧急数据，因此对于计算机之间的突发式的数据通信，分组交换显然更为合适些。

分组交换的缺点：

- 尽管分组交换比报文交换的传输时延少，但仍存在存储转发时延，而且其节点交换机必须具有更强的处理能力。
- 分组交换与报文交换一样，每个分组都要加上源、目的地址和分组编号等信息，使传送的信息量增大 5% ~ 10%，这在一定程度上降低了通信效率，增加了处理的时间，使控制复杂、时延增加。
- 当分组交换采用数据报服务时，可能出现失序、丢失或重复分组，分组到达目的节点时，要对分组按编号进行排序等工作，增加了麻烦。若采用虚电路服务，虽无失序问题，但有呼叫建立、数据传输和虚电路释放三个过程。

总之，若要传送的数据量很大，且其传送时间远大于呼叫时间，则采用电路交换较为合适；当端到端的通路有很多段的链路组成时，采用分组交换传送数据较为合适。从提高整个网络的信道利用率上看，报文交换和分组交换优于电路交换，其中分组交换比报文交换的时延小，尤其适合计算机之间的突发式的数据通信。

1.7 VoIP

维基百科上是这样说的：IP 电话（Voice over Internet Protocol，VoIP，又称宽带电话或网络电话）是一种透过互联网或其他使用 IP 技术的网络来实现的新型电话通信。过去 IP 电话主要用在大型公司的内联网内，技术人员可以复用同一个网络提供数据及语音服务，除了简化管理，更可提高生产力。随着互联网日渐普及，以及跨境通信数量大幅飙升，IP 电话亦被应用在长途电话业务上。由于世界各主要大城市的通信公司竞争日益剧烈，以及各国电信相关法令松绑，IP 电话也开始应用于固网通信，因其具有的低通话成本、低建设成本、易扩充性及日渐优良化的通话质量等主要特点，被目前国际电信企业看成是传统电信业务的有力竞争者。更详细的内容参见维基百科上的“IP 电话”㊀。

目前，VoIP 呼叫控制协议主要有 SIP、H.323、MGCP 与 H.248/MEGACO 等。

㊀ 参见 http://zh.wikipedia.org/wiki/IP 电话。

1.8 IMS

IMS[⊖]涉及的概念和名词术语相当多，本节将简单加以介绍，对此感兴趣的读者参考，也可以根据这里提到的关键词到网上搜索或查找相关书籍进行更深入的学习。其他读者可跳过本节。

1.8.1 什么是 IMS

IMS 的全称是 IP 多媒体子系统（IP Multimedia Subsystem），它是一个基于 IP 网提供语音及多媒体业务的网络体系架构。它最初是由 3G 标准化组织 3GPP[⊜]设计的，作为其 GSM 之后的未来移动网络远景目标的一部分。IMS 的最初的版本（3GPP R5）主要是给出了一种基于 GPRS 来实现 IP 多媒体业务的方法。在这个版本的基础上，3GPP、3GPP2 以及 TISPAN 进行了进一步的更新，以支持 GPRS 之外的（诸如 WLAN、CDMA2000 和固定等）其他接入网络。从目前来看，IMS 是独立于接入网技术的，尽管它与底层传输功能有着很多联系。

从另外一个角度看，IMS 实际上是 IP 网上的一个应用系统。IP 网的相关技术标准主要由 IETF 制定，包括应用层（如 Email（POP3、SMTP）、文件传输（FTP）、网页浏览（HTTP）等）的相关协议标准。IETF 负责制定了与实时应用（Real-time Applications）相关的协议标准，包括 SIP、RTP 等。IMS 使用的基本都是 IETF 相关的协议标准（SIP、Diameter 等），不同的是，ISM 在其基础上又进行了详细的操作性描述和增强，以便提供一种完整的、健壮的多媒体系统。这些操作性描述和增强为运营商控制、分责任、计费和安全提供了支持。

IP 多媒体的全套解决方案是由终端、GERAN（GSM EDGE Radio Access Network，GSM/EDGE 无线通信网络）或 UTRAN（UMTS Terrestrial Radio Access Network，UMTS 陆地无线接入网）、GPRS 核心网和 IP 多媒体核心网子系统的一些特殊的功能单元来支持的。这些功能单元包括呼叫会话控制功能（CSCF）、媒体网关控制功能（MGCF）、IP 多媒体网关功能（IM-MGW）、多媒体资源功能控制器（MRFC）、多媒体资源功能处理器（MRFP）、签约定位功能（SLF）、出口网关控制功能（BGCF）、应用服务器（AS）、信令网关功能（SGW）等。

IMS 网元众多，其核心网络基本架构如图 1-12 所示。

1.8.2 IMS 的特点

IMS 具有以下特点：

- ❑ 采用 SIP 作为呼叫控制协议。基于 SIP 协议实现了呼叫控制和业务控制的分离，并增强了多媒体支持能力。

⊖ 参见 http://zh.wikipedia.org/zh-cn/IP 多媒体子系统。

⊜ 第三代合作伙伴计划（3rd Generation Partnership Project）是一个成立于 1998 年 12 月的标准化机构。目前其成员包括欧洲的 ETSI、日本的 ARIB 和 TTC、中国的 CCSA、韩国的 TTA 和北美洲的 ATIS，详见 https://zh.wikipedia.org/wiki/3GPP。

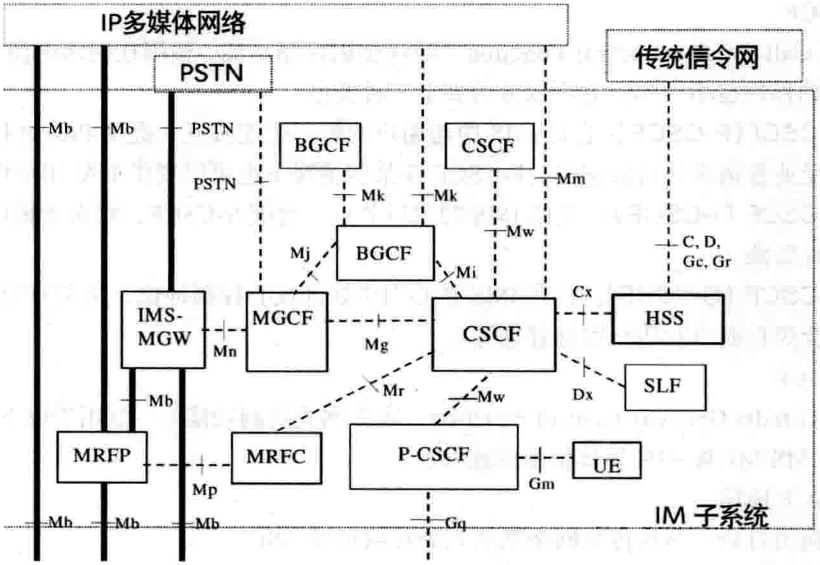


图 1-12　IMS 基本架构

❑ 支持 Diameter 协议。Diameter 是 IETF 开发的协议，用于认证、授权和计费（Authentication、Authorization、Accouting，AAA）。

❑ 采用归属控制方式。对于移动用户而言，通过归属控制，即使用户漫游到外地，也可以享受到与归属地同样的服务。

❑ 采用接入无关性。提供优越的融合特性，核心功能与具体接入技术无关。

❑ 业务、控制、承载层完全分离。IMS 进一步发扬了 NGN 软交换结构中业务与控制分离、控制与承载分离的思想，与软交换相比其进行了更充分的网络解耦，网络结构更加清晰合理，同时不同类型网络的解耦也为网络在不同层次上的重新聚合创造了条件。这种重新聚合，就是网络新的融合的过程。

❑ 增强计费功能。通过 CCF（计费采集功能），可以支持更灵活的在线、离线计费。

❑ 增强多媒体业务。在增强多媒体业务这方面，主要体现在 Presence（呈现）、Messaging（短消息）、Conferencing（会议）、PoC（Push-to-talk over Cellular，基于移动网络、采用 VoIP 技术的集群对讲业务）、MBMS：Multimedia Broadcast Multicast Service（多媒体广播多播服务）等几个方面。

1.8.3　IMS 核心网元

IP 多媒体子系统像 CS 域（Circut Switched Domain，用于向用户提供电路型业务连接）、PS 域（Packet Switched Domain，用于向用户提供分组型业务的连接）子系统一样，可以完成呼叫的发起、保持、释放等功能。另外，它还要对多媒体进行转换控制以及对多媒体业务提供支持，所以包含更多的功能实体来分别完成不同的功能。

（1）CSCF

CSCF（Call Session Control Function，呼叫会话控制功能）根据在网络中所处的位置的不同，承担的作用也不一样，它可以分为如下三种类型：

- **代理 CSCF(P-CSCF)**：它是 IMS 中与用户的第一个连接点，提供 Proxy(代理）功能，即接受业务请求并转发它们。P-CSCF 在某些情况下也可以提供 UA(用户代理）功能。
- **问询 CSCF（I-CSCF）**：类似 IMS 的关口节点，分配 S-CSCF、路由查询以及 IMS 域间拓扑隐藏。
- **服务 CSCF（S-CSCF）**：它在 IMS 核心网中处理核心控制地位，负责对 UE 的注册鉴权、会议控制以及用户数据管理等。

（2）MGCF

MGCF（Media Gateway Control Function，媒体网关控制功能）一般用于以下场景：

- 控制 IMS-MGW 中的媒体信道的连接。
- 与 CSCF 通信。
- 根据路由号码，为从传统网络来的入局呼叫选择 CSCF。
- 执行 ISUP 协议和 IMS 呼叫控制协议间的转换。

（3）IM-MGW

一个 IM-MGW（IP Multimedia-Media Gateway Function，多媒体网关功能）可以终止来自电路交换网的承载信道和来自分组网的媒体流（如 IP 网中的 RTP 流）。IM-MGW 可以支持媒体转换、承载控制和负荷处理（例如，多媒体信号编解码器、回声消除器、会议桥等）。它包含如下功能：

- 通过与 MGCF 交互来进行资源控制。
- 拥有并维护回声消除器等资源。
- 可能需要多媒体数字信号编、解码器。

IMS-MGW 要提供必要的资源来支持 UMTS/GSM 媒体传输，还需要对 H.248 协议进行进一步的调整来支持额外的多媒体数字编、解码器等。

（4）MRF

MRF（Multimedia Resource Function，多媒体资源功能）分成两部分，包括 MRFC（Multimedia Resource Function Controller，多媒体资源功能控制器）和 MRFP（Multimedia Resource Function Processor，多媒体资源功能处理器）。

- MRFC 的主要功能：控制 MFP 中的媒体流资源；翻译来自 AS 和 S-CSCF 的信息（会话标志符等），并相应地对 MRFP 进行控制；产生计费记录。
- MRFP 的主要功能：控制 Mb 接口点的承载；提供 MRFC 需要的资源，混合输入媒体流（如用于多方会议），发出多媒体流（如用于多媒体广播），处理多媒体流（如语音编码转换、媒体分析）等。

（5）SLF

在会话建立期间，被 I-CSCF 查询，SLF（Subscription Locator Function，签约定位功能）

向 I-CSCF 提供存储用户具体数据的 HSS 的名字；通过 Dx 接口来接入 IMS。在单一的 HSS 环境中，并不需要 SLF。

（6）HSS

HSS（Home Subscriber Server，归属用户服务器功能）是一个数据库实体，它用于在归属网络中保存用户的签约信息，包括基本标志、路由信息及业务签约信息等。HSS 中保存的主要信息包括：

- IMS 用户标识（包括公有及私有标志）：号码地址信息。
- IMS 用户安全上下文：用户网络接入认证密钥信息、漫游限制信息等。
- IMS 用户的路由信息：HSS 支持用户注册，并且存储用户的位置信息。
- IMS 用户的业务签约信息：包括其他 AS 增值业务数据。

（7）BGCF

BGCF（Breakout Gateway Control Function，出口网关控制功能）用于选择与 PSTN（或 CS 域）接口点相连的网络。如果 BGCF 发现自己所在的网络与接口点相连，那么 BGCF 就选择一个 MGCF，该 MGCF 负责与 PSTN（或 CS 域）的交互。如果接口点在另一个网络，那么 BGCF 就把会话信令转发给另一个网络的 BGCF。BGCF 在选择与 PSTN 相连的网络的时候，会利用收到的其他协议的信息和管理信息。BGCF 的主要功能如下：

- 收到 S-CSCF 请求后，为呼叫选择一个适当的 PSTN（或 CS 域）接口点。
- 选择一个与 PSGN（或 CS 域）相连的网络。如果本网络没有与 PSTN 相连，那么 BGCF 就把 SIP 信令转发给与 PSTN（或 CS 域）相连的网络的 BGCF。
- 在与 PSTN（或 CS 域）相连的网络中，选择一个 MGCF，并把 SIP 信令转发给 MGCF。
- 生成计费记录。

（8）SGW

SGW（Singnalling Gateway Function，信令网关功能）完成传输层的信令转换，在基于 SS7 的信令与基于 IP 的信令之间转换（也就是在 Sigtran SCTP/IP 和 SS7 MTP 之间进行转换）。SGW 不对应用层的消息进行解释，但必须对底层的 SCCP 或 SCTP 消息进行解释来保证信令的正确路由。

（9）AS

在 IMS 系统中，实现了业务与控制的完全分离，所有的具体业务都是通过应用服务器（Application Server，AS）来提供的。应用服务器通过一种称为开放服务架构（Open Service Architecture，OSA）的方式引入了 Internet 上应用的开发模式，为 IT 应用与电信网的融合奠定了技术基础。AS 与 CSCF 之间使用 SIP 协议通信。对于不同的服务，AS 可以选择不同的 SIP 模式，如 SIP 代理模式、SIP 用户代理（UA - User agent）模式和 SIP B2BUA 模式。AS 可以设置在 IMS 本网内，也可以设置在外部的第三方网络中。如果位于本网，它还可以利用 Sh 或 Si 接口查询 HSS。

一般来说，AS 包含以下三类功能与实体：

- SIP AS（Application Server）：基于 SIP 的应用服务器，负责提供 IMS 的具体服务。

SIP AS 和 S-CSCF 之间直接利用 SIP 及其扩展的呼叫信令协议，因此不需要进行呼叫信令协议之间的转换工作。另外由于基于 SIP 可以非常方便地实现语音、数据以及视频等多媒体类的会话，因此 SIP AS 可以高效率地提供各种新型的融合业务。

- IM-SSF（IP Multimedia Service Switching Function）：IP 多媒体交换功能实体，它作为 SIP 和智能网的 CAP（CAMEL[⊖] Application Part, CAMEL 应用部分）之间的接口，为 IMS 用户提供增值业务。可以位于用户归属网，也可以由第三方提供，主要用于处理 IMS 发来的 SIP 会话、发起 SIP 请求、发送计费信息给 CCF 和 OCS。
- OSA-SCS（Open Service Access-Service Capability Server）：SIP 和 OSA 框架之间的接口。SCS 实际上是负责 API 具体实现的功能实体，它与核心网络元素（如 HLR、MSC、SSP 等）进行交互。这样，一个 SCS 服务程序就相当于一个进入核心网络的一个代理或一个网关。

1.8.4 SIP 协议的参考点

IMS 网络中使用 SIP 协议的主要参考点如表 1-1 所示。

表 1-1 SIP 主要参考点

参考点	连接实体
Gm	UE、P-SCSF
Mw	P-CSCF、I-CSCF、S-CSCF
ISC	I-CSCF、S-CSCF、AS
Mg	I-CSCF、MGCF
Mi	S-CSCF、BDCF
Mj	BGCF、MGCF
Mk	BGCF、BGCF
Mr	S-CSCF、MRFC
Mp	MRFC、MRFP
Mn	MGCF、IM-MGW
Cx	CSCF、HSS
Dx	CSCF、SLF
Sh	HSS、SIP AS、OSA-SCS
Si	HSS、IM-SSF
Ut	UE、SIP AS

⊖ Customised Applications for Mobile networks Enhanced Logic，即移动网络定制应用增强逻辑服务器，它扩展了智能网（IN）提供给移动环境的业务范围，使移动网络能更容易地实现基于智能网的增值业务。参考：https://en.wikipedia.org/wiki/CAMEL。

1.9　小结

本章介绍了通信系统的一些历史故事和背景知识，其中 PSTN 起源部分简单介绍了交换机的起源及历史演变过程。科技的发展带来了技术的变革，但从另一方面讲，这些变化不仅是技术驱动的，更多的是在社会的发展过程中改变了人们的生产生活方式，又影响并推动了技术的发展。了解这些背景知识对于理解交换系统的各种功能会有很大的帮助。同时，读者也可以延伸思考一下：在这些技术发展的历程中，用户、运营商、设备制造商又各自扮演了哪些角色；他们各自得到哪些好处；有哪些痛处；这些又是如何取得平衡的；等等。思考这些问题有助于把握技术的发展方向，从而更容易理解技术的实现方式和来龙去脉。

本章还重点讲了信令的知识，并对典型信令间的互通及对应关系给出了呼叫时序图。由于本书处处都有 SIP 信令的影子，因此，希望这些时序图能给读者一个直观的印象。对初学者而言，理解各种信令之间既有所不同的，又有相似的地方就可以了，对于实际互通的细节则没必要深究。当然，读了后面的章节后再回到本章肯定能加深对信令系统的了解。

另外，我们还专门介绍了 IMS 基本概念。虽然 IMS 不是我们研究的重点，而且 IMS 中太多的概念和术语也会让初学者摸不着头脑，但 IMS 本身作为 SIP 协议的最大用户仍有了解的价值。通过对后面章节的学习，读者就能更深入了解 FreeSWITCH 可以在 IMS 网络中充当什么角色，进而更容易理解通信网的总体架构和设计思想。当然，对于初学者来说粗略了解一下 IMS 部分即可，等学完后面章节再回过头来看会更有心得。

第2章 PSTN、PBX及呼叫中心业务

我们在第1章学习了PSTN和VoIP的基本概念和术语，在本章我们接着了解一下传统的电话网及交换设备。这些服务有的是读者已经熟悉的，有的可能没听说过。有一些传统业务在VoIP时代实现起来异常简单，而有一些业务可能已经不需要了。

为了更深入地理解这些业务，在本章我们也对一些基本的概念，如中继线、IP-PBX等加以深入介绍。

另外，考虑到相当一部分读者可能把FreeSWITCH应用到呼叫中心或相关业务中，在本章我们也对呼叫中心业务进行了简单的介绍，并简单讨论一下在FreeSWITCH中的实现思路。

2.1 PSTN业务

我们还是从传统的PSTN业务开始讲起。除了为用户提供基本的语音通话外，PSTN还能提供一些附加的业务，如叫醒服务、呼叫转移等，为人们的生产和生活提供方便。这些业务有的可以由运营商为用户设置，有的也可以由用户拨打某些特殊的号码自行激活和取消。下面我们分类来介绍一下。

2.1.1 POTS

POTS（Plain Old Telephone Service）即普通老式电话业务。POTS是沿用国外的叫法，在国内我们称之为新业务。当然，新业务这种叫法这还是沿用数年前的叫法。这些业务有的是收费的，有的是不收费的。它们的接入号码通常以“*”开头。古老的话机是转盘式的，使用脉冲方式拨号，只能拨0～9的号码，现在在现实生活中已很少用了，但大家从电影上经常能看到。现代的话机多为按键式，使双音频方式拨号，上面有0～9、*和#字等键。其中，*字键通常读作“星”[⊖]。有的话机上还有A、B、C、D键，但这些键很少用到。

⊖ 有些运营商的话务员也将“*”读作“米”，笔者觉得不是很通用。

另外有的话机上还有各种其他的功能键，如号码翻查、重拨等。在某些新业务中（如三方通话），会用到话机上的叉簧，快速拍一下可以给交换机传递相关信号，某些话机上有专门的 Flash 键、R 键或闪断键。下面仅列举几种典型的业务：

- ❑ 缩位拨号（Abbreviated dialing）。通过事先登记的代码代替长号码，可以减少记忆难度，加速拨号。比如，拨“**1”可以拨叫之前指定的号码（比如 12345678），该功能比较实用。
- ❑ 呼叫前转（Call Forwarding，有时也称呼叫转移）。分三种基本情况：无条件转移，即任何来电转移至事先登记的号码；遇忙转移，若被叫忙，则转移；无应答转移（或称久叫不应转移），若指定时间内无应答，则转移。其中大部分交换机上无条件转移的登记方式为“*57* 电话号码 #”，取消方式为“#57#”。登记成功后，所有到该话机的来电会转到所登记的电话号码。如在话机 A 上操作“*57*B#”，则所有对 A 的呼叫都会转移到 B 上。适用于将家里或办公室电话转移到手机上的情况。运营商也经常使用该功能作一些特殊的业务，如改号通知，即通过后台操作将某一号码转移至特定的语音平台，实现类似“您拨的电话已改号，新的号码是 XXX”的功能。
- ❑ 新转移方式。在移动电话及小灵通等出现后，又有了新的转移方式，如不在服务区转移等。
- ❑ 立即热线（Hotline）。拿起电话不用拨号即自动拨打某号码，方便拨号。笔者在北京某银行网点用过该项业务，拿起电话直接连接到他们的自助语音服务。
- ❑ 延迟热线（Delayed Hotline）。与立即热线差不多，区别是摘机后会延迟一段时间（如 5 秒）再自动拨号，在特殊场合有用。
- ❑ 呼叫等待（Call Waiting）。被叫忙时，主叫仍会听到正常的回铃音（或个性化的语音提示：请不要挂机，您拨打的电话正在通话中……），而交换机会通过特殊的提示音提示有新电话呼入，被叫可选择是否接听，或在两者间切换。
- ❑ 三方通话（Three Way, Conference Call）。通过比较复杂的操作实现三方通话，某些交换机支持最多五方的通话（会议电话），更多方的电话会议系统需要专门的平台。
- ❑ 来电显示（Caller ID Presentation）。就是在被叫话机上显示主叫方的电话号码。
- ❑ 呼出限制（Call Barring，又称密码限呼）。呼叫某类号码（如长途电话）前提示先输入密码，在电话费还是很贵的年代比较有用。该功能可以使用密码限制话机能否打长途等，也可以限制小孩乱打电话。
- ❑ 免打扰服务（Do Not Distrurb，DND）。登记该业务后，如果有来电，交换机会提示主叫用户被叫用户不想被打扰。不过，在实际应用中，直接拔掉电话线比登记这个容易多了。
- ❑ 叫醒服务（Alarm Call，又称闹钟服务）。登记后在相应的时间电话会振铃。在这个手机异常普及的年代相信一般人不会用这个功能了，但在酒店应用中还是非常普遍的。
- ❑ 遇忙回叫（Completion of Call to Busy Subscriber 或 Auto CallBack，CCBS）。如果被叫忙，则主叫可以按一个特殊的号码以登记该业务，待被叫空闲后双方话机会自动振

铃，接听后双方进行通话，省了好多重复拨叫的操作。不过，该业务一般限制主、被叫用户都在同一交换机上，因而实际应用意义不大（最早发明该业务时可能人们的交际范围不广，大家都在同一交换机上）。

2.1.2 商务业务

商务业务是由运营商提供的，主要是为企业用户服务，一般有以下几种。

（1）模拟中继线

模拟中继线又称为用户小交换机⊖，它主要提供号码连选功能。典型应用是提供一个总机号（又称引示号）及若干条中继线（实际上就是普通的电话线）。当有人拨总机号时，交换机会根据指定的策略选择一条空闲的中继线呼入。而用户端通常会接 PBX 设备，下设分机。当用户呼出时，通过用户端的 PBX 设备选择一条空闲的线路。用户可选择是否显示总机号。

（2）数字中继线

如果用户需要的中线数量较多，数字中继线能提供更稳定的服务，设备通常支持 2Mbit/s 的一号信令或 30B+D 的 ISDN 信令，有的设备也支持 7 号信令。

（3）虚拟网

虚拟网又称商务组（Business Call Group，BCG）或汇线通（Centrex）业务。虚拟网主要提供在无需用户端 PBX 设备的情况下，实现网内（组）电话互拨小号，通常小号间的通话是免费的，但要比普通电话多收部分月租费。

虚拟网与模拟中继线的区别是，它的每路电话都是直线，可以直接呼入呼出，但需要占用更多的 PSTN 号码资源。它与普通电话的区别是，网内可以互拨小号。

（4）立即计费

传统的 PSTN 需要通过额外的系统来计算通话费用，通常需要有一段时间的滞后。而立即计费主要用于酒店等需要立即计费的场合，通常使用 ISDN 信令配合用户端的话务台软件实现。

（5）VPN

VPN（Virtual Private Network) 的全称是虚拟专用网，有别于 Internet 上的 VPN。它主要是用在连接大型企业在不同城市的分支机构中，可实现公司内部互拨小号，有时也称为广义虚拟网。

2.1.3 其他增值业务

传统的语音业务所带来的收入比例越来越低，因此，各大运营商都纷纷推出基于数据库和计算机系统的各种增值业务以增加收入。这些业务包括预付费业务（电话卡类业务等）、800 业务、400 业务以及彩铃、电话秘书台（语音信箱）等。当前，受苹果商店和安卓市场的影响和启发，随着云和大数据时代的到来，大家也都意识到靠几个大厂家研发的产品和服

⊖ 注意，这里说的模块中继线并不是“中继线”本身，而是指一种功能，可以理解为模拟中继线功能或用户小交换机功能。

务是永远跟不上时代发展的，因而各大运营商也在积极研发和试点新型的能力开放平台，力求将沉睡在运营商网络中的通信能力、数据等通过 API 或其他接口方式提供给开发人员和用户，吸引更多的开发者，从而创造更丰富的应用，为社会创造更多的价值。

2.2 PBX 业务

PBX（Private Branch eXchange）的全称是专用小交换机。该设备一般安装在企业内部。PBX 的上端通过运营商提供的模拟或数字中继线连接到 PSTN，而下端则直接接企业内部的话机。

企业使用 PBX 的好处是可以自己控制内部呼叫，而且内部通话免费。它通常可以提供呼叫保持、自动选线、呼叫前转、呼叫转移等基本功能，比较高级的小交换机还可以提供自动总机、三方通话、语音信箱等功能。在此，我们仅就经常使用的呼叫转移业务和同组代答业务简述如下。

2.2.1 呼叫转移

企业用总机一般有人工话务员，话务员接听电话后可能会根据需要将来话转移到其他分机[⊖]。转移有两种，一种是称为盲转（Blind Transfer），即被叫一方不管三七二十一将来话转至第三方号码，至于第三方号码是否可用，是否有人接听，则全然不管；另一种称为协商转（Attended Transfer），被叫一方先通过一些操作将来话置于 Hold 状态，主叫一方听音乐，被叫一方呼叫第三方号码，第三方接听后，被叫可询问第三方是否愿意接听，然后再执行转移操作或挂机。关于这两种转移操作，我们在 15.1 节还会详细讨论。

2.2.2 同组代答

通过逻辑上将一些分机分配到一个组，组中其他电话振铃时，组内的任意人可以拿起话筒拨叫一个特殊号码将正在振铃的某一分机上的呼叫接到本机上来。在办公室人数较少又不想漏接电话的情况下可使用本业务。

2.3 PBX 与中继线

用户或企业 PBX 要想打通外面的电话，或者外面的电话需要打进来，需要走运营商提供的中继线，以接入到 PSTN 网上去。理解中继线的概念对于理解 PBX 以及 PSTN 是非常重要的，中继的接入方式决定了我们如何拨号，读者在学习中也可以思考一下为什么使用这

⊖ 与上一节所述的呼叫前转不同（虽然人们有时也将前转称为呼叫转移，但在本书中，我们认为前转和转移是不同的概念），“前转”来自英文 Call Forwarding，是由交换机实现的，在电话到达被叫用户之前就被转移到其他号码了；而“转移”来自英文的 Transfer，是被叫用户接听后，将来话转移到其他号码。所以，我们认为，“前转”是一项业务，而“转移”指一个动作。

种拨号方式，我们能如何配置以提供更好的用户体验等。因此，在这里我们单独拿出一节来说明。

下面我们以模拟中继线为例，通过一则故事来说明中继线与 PBX 的关系。

假设我们刚开了一家公司，需要 7 部电话，于是向运营商（在 PSTN 交换机上）申请了 7 条模拟中继线。前面已经指出，实际上就是 7 条普通的电话线，只是运营商在 PSTN 交换机端对我们这 7 条线（也可以解释为 7 个号码）做了特殊的数据设置，将其逻辑上分为一个组，并为该组设了一个总机号。我们有幸选到了一个很酷的号码——88888888（它可以是一个虚拟的号码，或者是其中某一条中继线的真实号码）。而其他的中继线号则可能是 44440001 ~ 44440007。现在，我们把这 7 条线都接上话机。如果有人呼叫 88888888，则 PSTN 交换机会从 7 条线中自动选择一条空闲的线路呼入，因此某个电话会就会振铃。如果有多个电话呼入，只要同时呼入的电话不超过 7 个，我们的电话就都有机会振铃，因而我们可以同时对外为 7 个人同时提供服务。一般来说，当有电话呼入时，交换机有两种选线策略——顺序选线和循环选线。所谓顺序选线，就是每次都从 44440001 开始，寻找一条空闲的线路进行呼入；而循环选线则是每次都从上一次呼叫的下一个开始选起，使用这种选线方式，每个话机接到的电话数会比较平均。公司安装的 7 部电话的结构示意如图 2-1 所示。

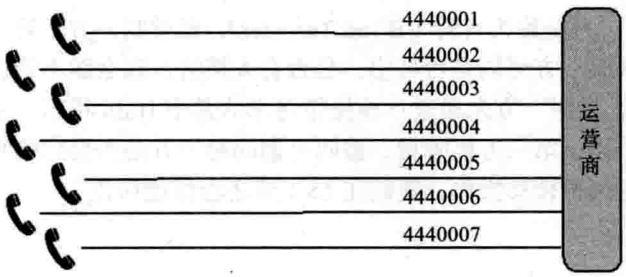


图 2-1 运营商为公司安装了 7 部电话

为维护企业形象，当有人呼出时，不管是从哪个分机呼出，都显示总机号 88888888。当然，也可以设置显示单线的号码（如 44440004），这个要在 PSTN 交换机端设置，一旦设置后，用户端不能动态更改。

一个月后，公司发展到 21 个人，因此需要 21 部电话。但由于一般不会出现所有人同时都在打电话的情况，故安装 21 条线有些浪费。因此我们买了一个小交换机，把原来的 7 条中继线接到小交换机的外线接口上，而把每个人的话机接到小交换机的内线口上，这样，每个人就都有了一个分机号，从 601 到 621，而 PSTN 端的配置不变，如图 2-2 所示。当客户打总机号时，PSTN 交换机仍然会选择一条线进入我们的小交换机，这时候，选线方式已经不像以前那样重要，因为现在是小交换机在接电话，对它来说，7 条线哪条都一样。就这样，小交换机接了电话，并播放“您好，欢迎致电某某公司，请直拨分机号，查号请拨 0……”，如果客户按某一分机号，则对应分机振铃，电话接通。

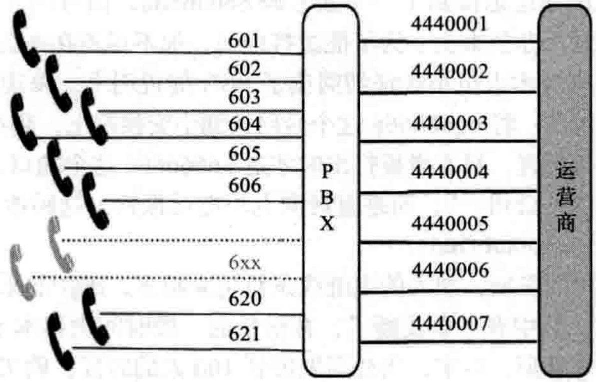


图 2-2 小交换机（7 条外线，21 条内线）

有了小交换机，内部通话就免费了。但出现了另外一个问题，就是如果拨打外线，则需要先拨一个特殊的数字，一般是 0 或 9。有的小交换机会送二次拨号音，即你拿起电话，听到小交换机的拨号音，拨了 0 之后，则听到外部 PSTN 交换机的拨号音，表明你可以拨打外线了。总之，小交换机会选择一条空闲的中继线对外呼叫。

上述例子中，21 : 7 称为集线比，即 3 : 1。集线比是由话务量决定的，如果同时通话的人数比较多，那我们可能会把中继线增加到 12 条，集线比就降为 21 : 12，约为 2 : 1 了。

即使增加了线路，也经常会遇到这样的情况：由于打进来的电话太多，占用了太多的线路，经常一个电话都打不出，因此，我们联系运营商，将中继线分为三组，其中 4 条只进不出，4 条只出不进，4 条能出能进。在电信术语中，分别叫做单出，单入和双向[⊖]，而北京联通则分别称为发专、受专和双向。当然，这种分配方式降低了总体线路的使用率，为此，我们把每个组都增加 1 条线，现在中继线总共达到 15 条。

又过了几天，有客户反映这样的情况，正常上班期间打电话经常无人接听，需要打好几遍；而同时，内部也有人反映往外打电话时有时拨 0 没反应，再试一次就好了。我们没有处理这种问题的经验，只好请教 PBX 专家，专家说可能是某条外线断了。因为，如果有一条线断了，当有电话呼入时，交换机仍会向主叫方送回铃音，跟被叫端没接话机是一样的。但到底是哪条线断了，却不好查。由于双方都是自动选线。我们只好将每条线都从小交换机上拔下来，接上话机试一试[⊜]，以确定是哪条线断了。

还算比较幸运，我们找到了断线的号码，联系运营商，很快修好了，把所有线路都插回小交换机，一切恢复正常。

⊖ 注意，对电信部门来说，出和入跟我们是相反的，因为我们（小交换机）的出，对应它们（PSTN 交换机）的入。

⊜ 某些小交换机也支持指定端口拨打功能，即在拨打真正号码前先拨几个特殊的数字，可以选择从指定的线路出去。实际上还有一种方法：我们知道，一般来说，每条单线的号码也都是可以呼入的，只是外人不知道而已，故只需要依次拨打所有的单线号码就可以知道是哪条线断了。当然，这依赖于 PSTN 交换机端的设置，有些城市（甚至同一城市不同的交换机都有不同的设置）默认设置单线是不允许呼入的。

几天后，老板又很幸运地搞到了一个新号码 66666666，该号码并未加入中继线组，而是直接扯了根线拉到老板办公桌上。为了能拨打内线，他不得不在办公桌上放两部电话，另一部专门打内线。后来技术人员小张仔细阅读了 PBX 的说明书，发现该小交换机功能还比较强，就进行了以下设置：将 66666666 这个号码接到小交换机上，仍给老板一个内线电话，同时在小交换机上进行设置，只有老板打出时才走 66666666 这个端口；而对于打入的电话，也不播放“欢迎致电 XX 公司…”，而是直接向老板电话振铃。这种拨入方式叫做 DID，即对内直接呼叫（Direct Inbound Dial）。

接下来，随着公司的发展，加入的中继线条数越来越多，维护起来更加复杂。比如，像我们刚才遇到的情况，其中有一条线断了，在很长的一段时间内根本不知道，即使知道了，要找到是哪条线也非常麻烦。后来，当公司发展到 100 人的时候，购买了新设备，并将模拟中继线换成了两条 E1 数字中继线，可同时支持 60 路通话。

公司发展一帆风顺，电话量也越来越多，公司有了很多分支机构，也有了更多客户，需要更复杂的语音菜单及更智能的电话分配策略，而更换专门的电话系统不仅价格昂贵，而且跟现有业务系统进行集成难度也很大。在综合考虑了多种解决方案以后，技术人员开始学习 FreeSWITCH……

2.4 IP-PBX [⊖]业务

在上一节中，我们最初买的模拟和数字小交换机是基于电路实现的，在这里我们将它们称为传统的 PBX。同时我们也欣喜地看到，我们的技术人员已经开始学习和研究 FreeSWITCH 了。FreeSWITCH 的默认配置就是一个家用或小型企业级的 PBX，它是由纯软件实现的，基于 IP 网进行通信，因而又称为 IP-PBX。

IP-PBX 首先是一个 PBX（Private Branch eXchange），它具有传统 PBX 的绝大部分功能。另外，由于使用了 IP 通信，它能通过 IP 网提供语音、视频以及即时消息通信。这些通信不仅可以在企业内部网上进行，也可以通过 Internet 在外网甚至 PSTN（Public Switched Telephone Network）电话间进行。

由于大部分 PBX 功能都是用软件实现的，因而实现起来成本相对来讲都非常廉价，并且非常易于增加新功能，如多方会议、使用 XML-RPC 等控制正在进行的通话、IVR、TTS/ASR（Text To Speech/Automatic Speech Recognition）、支持通过模拟或数字线路与 PSTN 网络对接，支持通过 SIP、IAX、H.323 或 Jingle（Google 对 XMPP 协议的扩展）以及其他协议与其他通信系统进行互联互通等。

与传统的 PBX 相比，IP-PBX 支持更多的新特性（或更易于支持某些特性）：

- ❑ 无限分机数量、无限自动话务台、无限语音信箱；
- ❑ 更易于通过 API 与其他应用系统（如 CRM 等）集成；

⊖ 本节大部分内容来自 Wikipedia: http://en.wikipedia.org/wiki/IP_PBX，http://zh.wikipedia.org/wiki/ 网络电话交换机。

- 支持远端电话分机、支持软电话；
- 支持高级用户接口，如电话跟随、统一消息、电话录音、语音邮箱、传真集成等；
- 根据时间路由、自定义路由规则；
- 支持从语音邮箱回拨、语音邮箱转 Email；
- 支持电话监听、耳语；
- 支持根据名字呼叫；
- 支持话务员控制台、拖拽转移、点击拨号等；
- 支持多人会议室；
- 支持高级 IVR（Interractive Voice Response）；
- 支持自动呼叫分配（Automatic Call Distribution，ACD）；

除了上面所述的这些特性外，IP-PBX 更易于部署，尤其是基于 IP 的通信更加廉价。但是，IP-PBX 并不是解决所有问题的良药，老技术向新技术过渡总要有些取舍。比如，在 IP 环境中，IP 电话终端更加智能，如摘机检测、挂机检测、收号等，这些原来由 PBX 或交换机实现的功能现在都在终端上实现了，因而在 PBX 上很难获得这些信息的细节。另外，当它与传统的 PBX 或 PSTN 网络对接时，还需要相应的 VoIP 网关来实现。当然，现在国内某些运营商也开始试验性地提供基于 IMS 或 SBC 的 SIP 中继，这样对接起来就方便多了。

2.5 呼叫中心

基于企业级的 PBX 和 IP-PBX 的通信还只是局限于基础的通信层。而随着企业规模的扩大及用户对服务要求的提高，企业更需要在业务逻辑和管理层方面为用户提供更好的服务。当这些服务可以通过远程电话支持的方式解决的情况下，一种称为呼叫中心的业务（Call Center）便产生了。

在呼叫中心中，有专门的话务员为客户提供服务。呼叫中心通常能同时处理大量的通话，并且为了给客户更好的服务体验，呼叫中心的通信系统通常通过技术手段与 CRM（Customer Ralationship Management，客户关系管理）系统集成。

本节我们简单介绍呼叫中心的基本概念及其在语音业务中的应用模式、技术发展情况以及 FreeSWITCH 在呼叫中心应用中的优势。

2.5.1 什么是呼叫中心

呼叫中心又称客户服务中心，它是一种基于 CTI 技术、充分利用通信网和计算机网的多项功能集成，并与企业连为一体的一个完整的综合信息服务系统，利用现有的各种先进的通信手段，高效地为客户提供高质量、高效率、全方位的服务。初看起来呼叫中心好像是企业在最外层加上一个服务层，实际上它不仅为外部用户，也为整个企业内部在管理、服务、调度、增值方面起到非常重要的统一协调作用⊖。

⊖ 摘自百度百科：http://baike.baidu.com/view/4061981.htm。

通俗地讲，呼叫中心是企业或机构建立的以电话为主要手段，为客户提供服务与沟通的部门组织及信息系统。百姓生活中常见的 110、119、120 这些应急服务电话，及电信运营商的客服电话、电话银行等都是呼叫中心的具体应用。在呼叫中心中通常是由座席代表通过电话为客户提供相应的服务与沟通。根据呼叫中心业务量不同，可以同时处理的电话量和座席代表的人数也有所不同。较小规模的呼叫中心只有几个人，大规模的呼叫中心会达到几千人。

2.5.2 呼叫中心的历史

1956 年美国泛美航空公司建成了世界上第一家呼叫中心，在 20 世纪 80 年代，呼叫中心在欧美等发达国家的电信企业、航空公司、商业银行等领域得到了广泛的应用。20 世纪 90 年代中后期，随着中国经济的发展，呼叫中心概念被引入国内。今天，呼叫中心在家电企业、邮电、银行、航空、铁路、保险、股票、房地产、旅游、公共安全等众多的行业间搭建起了企业与客户、政府与百姓之间的一座桥梁，与百姓的日常生活息息相关。

呼叫中心技术的发展可以分为以下几个阶段。

（1）第一代呼叫中心

第一代呼叫中心最早出现在民航服务领域，用于接受旅客的机票预订业务。第一代呼叫中心的系统主要在早期 PBX 的基础上增加了电话排队功能，那时甚至不能称为呼叫中心，而称为热线电话，其全部服务由人工完成。

（2）第二代呼叫中心

IVR（Interactive Voice Responce，交互式语音应答）系统的出现，标志着第二代呼叫中心的开始。在呼叫中心中利用 IVR 系统可以将大部分常见问题交由系统设备通过语音播放、DTMF（双音多频，电话机上面的数字按键所发出的频率）按键交互解决。例如我们在日常生活中常用的 121121 天气预报、117 报时电话，通过电话银行进行余额查询、转账等业务都是通过 IVR 系统自动实现的。在第二代呼叫中心中，IVR 系统的大量使用，可以大大减少人工业务的受理数量和人工座席的工作强度，同时可以为客户提供 7 × 24 小时全天候、不间断的服务。

（3）第三代呼叫中心

随着计算机技术的发展，CTI（Computer Telephony Integration，计算机电话集成）技术的诞生与应用，标志着第三代呼叫中心时代的开始。CTI 技术实现了电话交换机系统与计算机系统的集成，即实现了语音与数据的同步。客户信息与资料采用数据库方式存储，座席代表可以在处理电话服务的同时可以从计算机系统中调取和修改客户信息数据，为客户提供个性化的服务。CTI 技术的使用，推动了呼叫中心更大范围地使用。与此同时，呼叫中心中出现了专门用于电话录音的录音设备，对座席代表与客户的通话进行录音、存储和查询。

相比之前的呼叫中心系统，CTI 技术的使用使得呼叫中心大部分功能实现了自动化。从客户电话接入到最终问题的解决，整个过程被完整地记录了下来。

（4）第四代呼叫中心

前三代呼叫中心均是以电话为主要的服务渠道。在 2000 年，伴随着互联网以及移动通

信的发展与普及，将电子邮件、互联网、手机短信等渠道接入呼叫中心，成为第四代呼叫中心的标志。第四代呼叫中心也称为多媒体呼叫中心或联络中心（Contact Center）[⊖]。它相对传统呼叫中心来说接入渠道丰富，同时引入了多渠道接入与多渠道统一排队等概念。

（5）下一代呼叫中心

目前，已经有厂商提出了第五代呼叫中心的概念。下一代呼叫中心的发展方向是在第四代多媒体呼叫中心的基础上，更多地融入了依托于互联网技术的媒体渠道与沟通渠道。例如：社交网络、社交媒体（如微博、微信等媒体渠道），依托于互联网的文本交谈、网上音频、网上视频等沟通渠道。

2.5.3 呼叫中心的分类

对于呼叫中心的分类可以有多种维度，如技术架构、呼叫类型、建设规模、功能、使用性质、呼叫方式、部署地点等。从呼叫方式上讲，它主要分为外呼型呼叫中心（如电话营销）、客服型呼叫中心（如客户服务）以及混合型呼叫中心（营销和客服融合）。除此之外，在这里，我们仅针对技术架构进行简单讨论。

按照呼叫中心系统所采用的技术架构的不同，呼叫中心可以分为交换机、板卡、软交换（IPCC）三种类型。

1. 交换机类型的呼叫中心

交换机类型的呼叫中心的呼叫中心系统是在交换机基础上构建而成的。呼叫中心系统平台主要由交换机系统、CTI 中间件、IVR 系统、录音系统组成。用于呼叫中心系统平台常见的交换机系统有：Avaya、西门子、华为、阿尔卡特 – 朗讯等。图 2-3 是一个典型交换机类型的呼叫中心平台系统架构图。

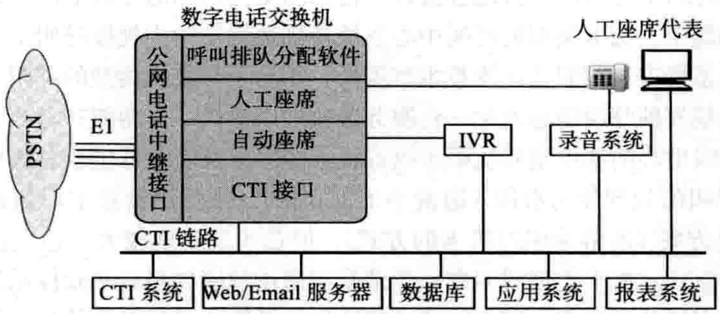


图 2-3 交换机类型的呼叫中心架构示意图

交换机类型的呼叫中心系统平台语音接入平台采用的是成熟稳定的交换机系统，采用专用硬件设备，具有产品成熟稳定、接入能力强、处理能力大等特点，适用于大中规模呼叫

⊖ Call Center 与 Contact Center 的缩写都是“CC”，由于 Contact Center 涵盖的业务面更广，因而现代大多数人倾向于使用联络中心这一概念。但呼叫中心这一名词已深入人心，因此，本书中仍然使用呼叫中心。

中心的建设。同时，交换机类型的呼叫中心系统平台具有系统架构复杂（需要由多个系统组成，通常会由多个厂家提供），安装部署及运维难度大，建设成本高等缺点。

2. 板卡类型的呼叫中心

板卡类型的呼叫中心系统平台语音接入平台采用语音板卡实现。语音板卡按照功能分为中继卡和用户卡两种类型，分别用于中继线的接入和座席端电话终端设备的连接。板卡类型的呼叫中心系统配合会议卡和传真卡还可以实现电话会议和电子传真的功能。随着 VoIP 技术的普及，一些语音板卡厂商也推出了自己的 IP 板卡，可以支持 IP 座席或 IP 中继的接入。图 2-4 是一个典型板卡类型的呼叫中心平台系统架构图。

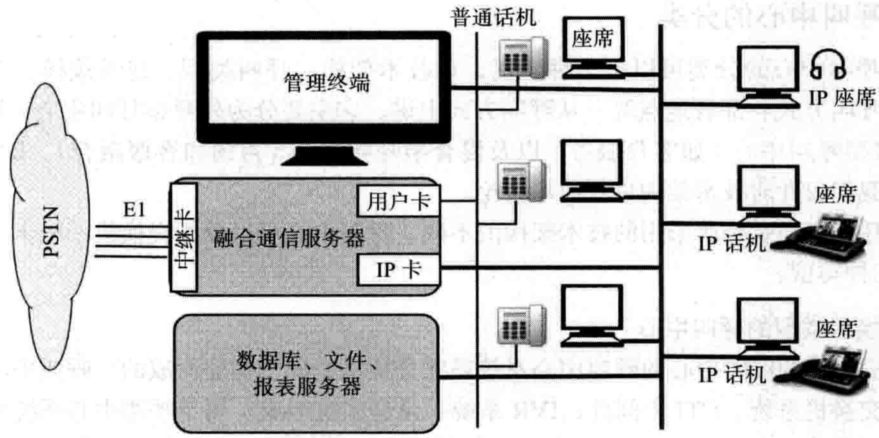


图 2-4 板卡类型的呼叫中心架构示意图

板卡类型的呼叫中心系统平台语音接入平台采用工控机加语音板卡的方式实现，其最大特点是系统结构简单。板卡类型的呼叫中心系统特别适用于中小规模呼叫中心的建设。板卡类型的呼叫中心系统安装部署及运维技术难度低，相对于交换机类型的呼叫中心系统其建设成本低。对于小规模的呼叫中心通常一台服务器就可以提供一套功能齐全的呼叫中心系统。

板卡类型呼叫中心的缺点是其受限于语音板卡的容量及服务器主板总线槽位数量，单台服务器对语音呼叫的处理能力有限（通常小于 120 路）。虽然语音板卡厂商也提供多台设备堆叠扩展的解决方案（通常采用过机卡的方式），但是实现难度较大、稳定性差，因而很少有厂商支持这种方式。板卡类型呼叫中心通常采用通用的硬件平台和软件系统，所有呼叫过程需要上层应用程序开发实现，其系统的功能和稳定性取决于开发商的技术能力和产品成熟度。相对于交换机厂商几十年的产品成熟度，国内厂商所提供的板卡类型呼叫中心解决方案在系统的稳定性和功能上还有很大的差距。

3. 软交换类型的呼叫中心

随着基于 VoIP 的软交换技术的发展，特别是一些优秀的开源软交换项目的出现，在呼叫中心系统建设方面出现了以软交换技术为核心的呼叫中心系统。软交换类型的呼叫中心不

仅解决了板卡类型呼叫中心接入能力受限于硬件板卡的问题，同时还具备板卡类型呼叫中心系统结构简单、部署灵活、低成本的优势。除 E1 接入外，现在有的运营商也提供 SIP 线路，较交换服务器也可以绕过 E1 网关而直接通过运营商的 SBC 接入 PSTN。图 2-5 是一个典型软交换类型呼叫中心的系统架构图。

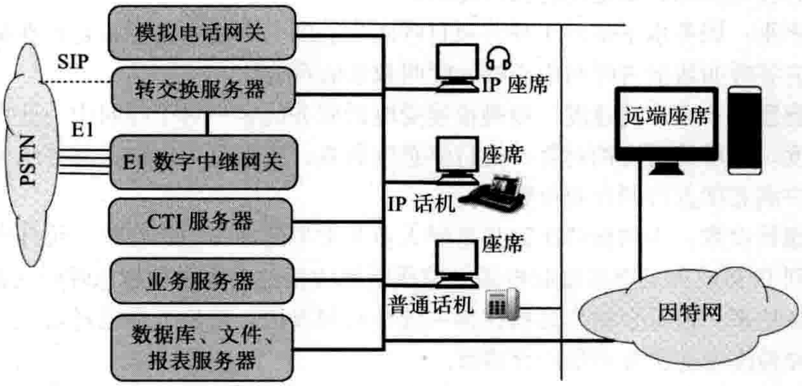


图 2-5 软交换类型的呼叫中心架构示意图

软交换类型的呼叫中心系统基于 VoIP 技术，具备先天的分布式部署的优势，特别适用于座席职场分散、接入分散的呼叫中心。软交换类型呼叫中心的系统容量可以通过多台服务器集群的方式平滑扩展，可以满足大型呼叫中心对于容量和性能的要求，是呼叫中心技术的发展方向。但是，软交换类型呼叫中心系统的性能和稳定性与呼叫中心厂商自身的研发能力有很大的关系。早期大多数软交换类型呼叫中心都是基于 Asterisk 或其他开源项目演变而来，受限于这些项目早期版本自身的问题，在市场上留下了性能差、不稳定的印象。这不是软交换技术本身的问题，而更多的是呼叫中心厂商自身技术能力的问题，它们大多采用“拿来主义”、“站在巨人肩膀上”的策略，在通信底层不愿进行太多的研发和投入，另外即使自己解决了某些问题一般也不会向开源社区公开，因而对开源软件的生态环境没有太大的贡献。

2.5.4 呼叫中心的主要技术指标

呼叫中心是一种劳动密集型产业。一个呼叫中心少则几个人，多则成百上千人。为了对呼叫中心工作人员进行有效管理，通常会通过一些量化技术指标来制定各种 KPI（Key Performance Indications，即关键绩效指标），作为衡量座席代表工作指标。

每个呼叫中心会根据其具体的业务类型、企业要求来制定自己的 KPI 指标，其数量和标准也不尽相同。常见的 KPI 指标有接通率、呼入项目占有率、呼出项目工作效率、服务水平、客户满意度、平均振铃次数、监听合格率、一次性解决问题率等。一般来说，这些指标的得分越高，绩效就越好。

- **接通率**：对于呼入业务类型的呼叫中心，接通率是指 IVR 终极服务单元的接通量与人工座席接通量之和与进入呼叫中的呼叫量之比；对于呼出业务类型的呼叫中心，接

通率是指座席呼出电话后接通量与呼出电话总量之比。

- **呼入项目占有率**：呼入项目占有率是指在某一统计时间段内，人工座席处理电话的总时长与实际登录系统时长的比率。
- **呼出项目工作效率**：呼出项目工作效率是指在某一统计时间段内，人工座席处理电话的总时长与实际登录系统时长的比率。
- **服务水平**：服务水平是对于呼入项目呼叫中心的一个技术指标，是指在某一统计时间段内应答呼叫数量占呼叫中心接入呼叫数量的百分比。
- **客户满意度**：客户满意度一般是指接受电话服务的客户对于呼叫中心所提供服务的满意程度。一般通过定期对客户进行满意度调查，或在每一次电话服务结束后系统自动对客户满意度进行调查获得数据。
- **平均振铃次数**：平均振铃次数是指呼入业务类型的呼叫中心在某一统计时间段内，客户听到 IVR 或人工座席接起电话之前所听到的振铃次数之和与总呼叫次数的比值。
- **监听合格率**：监听合格率是指在某一统计时间段内，质检人员通过监控、电话录音等手段抽检座席的服务质量的合格率。
- **一次性解决问题率**：一次性解决问题率是在某一统计时间段内，不需要客户再次拨打呼叫中心电话也不需要座席员回拨或转接电话就可以解决客户所提出问题的电话量所占座席员接起电话量总数的比率。

2.5.5 CTI 中间件

在 2.5.2 节我们讲到，CTI 中间件是第三代呼叫中心的重要标志。那么，什么是 CTI 中间件呢？ CTI（Computer Telephony Integration）的字面意思为：计算机电话集成。早先的电话交换机（即 PBX）是一个独立、封闭的系统设备。随着计算机技术的发展和呼叫中心需求的提出，交换机设备厂商开始考虑为交换机增加一个可以受计算机系统控制的接口，由计算机系统通过某种协议获取交换机用户话机的状态信息以及对呼叫的控制命令。这种连接和控制接口称为 CTI-Link。

交换机的厂商繁多，不同交换机厂商所提供的 CTI-Link 的接口协议不尽相同，常见的有北电网络的 Merisian Link、Avaya 的 ASAI/TSAPI 以及欧洲交换机厂商所普遍遵循的 ECMA（欧洲计算机制造协会）提出的 CSTA 标准。因此，要针对每一个交换机厂商的 CTI-Link 协议进行开发是一件很繁杂的工作，因此人们提出了 CTI 中间件的概念。CTI 中间件在下层通过对各种 CTI-Link 协议的包装和抽象，屏蔽了各种交换机的不同，在上层为呼叫中心业务软件开发人员提供统一的 API 开发接口，这样开发人员开发的程序不仅能支持丰富的业务逻辑，还能适用于各种不同的交换机，增加了系统的灵活性，也提高了开发效率。同时 CTI 作为一种中间件产品，也一直长盛不衰。图 2-6 是 CTI 中间件在呼叫中心系统中的位置。

早期比较著名的 CTI 中间件有 CT-CONNECT、Genesys、Quintus 等。随着呼叫中心技术的发展，CTI 中间件的功能也在不断丰富。CTI 中间件由最初单纯的 CTI-Link 协议转换

和标准化的单一功能发展成为涵盖多渠道统一接入、智能 ACD 呼叫分配、智能预测外呼、现场监控工具、数据统计分析等多个功能模块的系统平台。

在互联网尤其是移动互联网技术飞速发展的今天，人们对电话与互联网的集成又提出了新的要求，因而各厂商也增加了一些面向互联网的接口。但是，传统的 CTI 技术由于其产生的年代及其本身的基础架构的局限性，一般来讲是不适合互联网的。而且，随着使用 VoIP 软交换设备替代传统的硬件交换机，以及运营商网络由传统的 PSTN 向 IMS 及 LTE 的发展，理论上讲，原来的以电路交换为基础的电话系统现在也都变成了计算机了，因而计算机－电话系统集成也就失去了它原本的意义。现代的 VoIP 电话系统直接可以提供更现代、更开放的集成接口，SIP、3GPP、Webservice、REST、大规模并发和集群等新型的协议和开发部署方式由于其更易于与互联网集成而受开发者青睐，因而，CTI 领域也必将迎来一场新的革命。

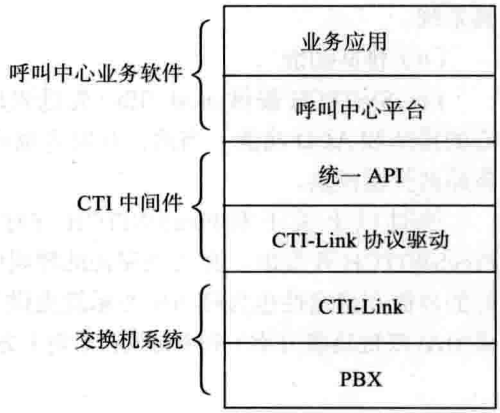


图 2-6　CTI 在呼叫中心系统中的位置

2.5.6　FreeSWITCH 在呼叫中心的应用

我们可以使用 FreeSWITCH 项目构建一个软交换类型的呼叫中心系统。本小节我们来看一下 FreeSWITCH 中和呼叫中心相关的几个主要功能。

（1）语音交换功能

FreeSWITCH 首先是一个 IP-PBX，具备类似交换机的语音交换功能。通过路由设置或程序控制，通过 FreeSWITCH 可以将客户拨打进的呼叫分配到指定的座席代表所在的话机终端。同样，座席代表也可使用话机终端通过 FreeSWITCH 发起呼叫，拨打客户的固定电话或移动电话。语音交换功能是构成呼叫中心接入平台最基本的功能。

（2）媒体处理功能

FreeSWITCH 具备媒体处理功能，可以进行录音、放音、DTMF 按键分析、产生 DTMF 按键的操作。利用 FreeSWITCH 的媒体处理功能可以实现呼叫中心系统中的 IVR 系统和录音系统的功能。

（3）媒体监播功能

FreeSWITCH 具备媒体监播功能。利用 FreeSWITCH 的媒体监播功能，可以实现呼叫中心所需的监听、强插、耳语等功能。

（4）电话会议功能

在某类呼叫中心业务中（如：电话外语翻译），需要多方参与通话时可以使用 FreeSWITCH 所提供的电话会议功能实现。

（5）电子传真功能

FreeSWITCH 支持收、发电子传真的功能，可以完全替代传统基于传真卡的语音板卡传真系统。

（6）排队功能

FreeSWITCH 提供 mod_fifo（先进先出模块）以及 mod_callcenter 模块用于实现呼叫中心的排队和 ACD 功能。当然，开发者也可以利用 FreeSWITCH 所提供的接口自己完成排队策略的控制模块。

通过以上关于对 FreeSWITCH 与呼叫中心相关功能的介绍我们可以发现，使用 FreeSWITCH 开发出一套功能完善的呼叫中心系统是完全没问题的。同时，FreeSWITCH 良好的性能和稳定性也为呼叫中心系统提供了可靠的保障。对于大规模呼叫中心的应用可以提供 HA（双机热备方案）和 Cluster（集群）方案来解决对于高可用性和大容量呼叫中心的需求。

2.6 小结

本章着重介绍了传统的 PSTN 网络和 PBX 系统所能实现的基本业务和增值业务。其中有一些业务是比较小众的，可能大部分人一生也不会用到；而很大一部分业务是在我们的工作和生活中常常用到的，只是可能没怎么在意。当然，对于这个领域的从业者来讲，这些都应该是非常熟悉的。但无论如何，在学习 FreeSWITCH 的过程中会涉及这些业务的方方面面。在本章，既有简单的罗列，也有生动的故事，目的就是带领对业务不熟悉的读者循序渐进地了解这些业务，对熟悉业务的读者也统一一下思想。读者在学习中不妨也深入思考一下：这些业务为什么会出现？它们给人们带来了怎样的便利？各种业务的出现都是为了解决谁的问题，是主叫用户、被叫用户，还是运营商？

本章也简单介绍了从 PBX 到 IP-PBX 的演变。值得一提的是，IP-PBX 能提供更丰富、灵活的功能，但有些人可能更喜欢传统的 PBX 简单。而且 IP-PBX 为兼容旧的设备和用户使用习惯往往采取一些妥协的办法，因而不一定能充分发挥其优势。旧技术到新技术的演变总要有一个过程，用户从认识、接受、慢慢习惯到熟练应用需要一个过程，各厂商的推动、研发以及利益方面的考虑也需要时间。新技术不是解决所有问题的良药，总有人会怀旧，但历史的车轮永远是向前的，新技术取代旧技术肯定是一个不可逆转的趋势。

最后，本章也用了大量的篇幅讲了呼叫中心。并不是因为呼叫中心这一概念有多重要，而是从它的历史和发展中可以看出交换机技术的发展以及进步，业务层五花八门的需求也将交换机的性能和功能都发挥到了极致。在呼叫中心部分我们也着重讲了 CTI 的概念，以及其将面临的挑战和革命。

总之，通过这两章的学习，我们已经了解了一些基础知识，端正了思想，统一了认识。接下来，便可以大步进入我们的 FreeSWITCH 之旅了。

第 3 章

初识 FreeSWITCH

在前面几章，我们用了很大的篇幅介绍了电话通信的背景和基础知识，以及电信业务的知识。对于刚刚跨入通信（或电信）领域的读者来说，熟悉这些背景知识以及里面提到的各种名词术语，有助于理解后面要学到的知识。通信领域涉及的面非常广泛，可以说，里面的很多术语或知识点单独拿出来都可以写成一章或一本书。我们本书的重点是 FreeSWITCH，因此从本章开始，我们正式进入 FreeSWITCH 的学习。学习本书的好处在于，即使你对前两章的内容不是很了解，也可以通过对 FreeSWITCH 的学习去反过来理解前面的知识。

在本章，我们将先讲解 FreeSWITCH 的基本概念，然后通过安装和简单的配置做成一个实际可用的 PBX 系统，并进行电话注册和拨打测试。目的是先给读者一个宏观、快速的体验，然后再在后面的章节中逐步强化和深入。读者可以循序渐进，一步一步成长为 FreeSWITCH 领域的高手。

3.1 什么是 FreeSWITCH？

什么是 FreeSWITCH？这一问题恐怕是初次见到本书的人首先要问的。很遗憾，我们一直到本章才回答这个问题。其实好多人问这一问题，并不是期望我们能给 FreeSWITCH 来下一个准确的定义，而是想知道，它到底能做什么。下面我们就来看一下 FreeSWITCH 的概念和功能。

3.1.1 FreeSWITCH 的概念

FreeSWITCH 是一个开源的电话交换平台。官方给它的定义是——世界上第一个跨平台的、伸缩性极好的、免费的、多协议的电话软交换平台[⊖]。由这个定义我们可以得出以下几点：

- FreeSWITCH 是**跨平台**的。它能原生地运行于 Windows、Max OS X、Linux、BSD 及

⊖ The World's First Cross-Platform Scalable FREE Multi-Protocol Soft Switch - http://www.freeswitch.org。

Solaris 等诸多 32/64 位平台（甚至，也有人成功地将它应用于 Linksys NLS2 平台及 Raspberry Pi 上[1]）。

- FreeSWITCH 具有很强的**可伸缩性**。FreeSWITCH 从一个简单的软电话客户端到运营商用级的软交换设备几乎无所不能。
- FreeSWITCH 是免费的。它采用 MPL 1.1 [2]协议授权，意味着任何人都可以免费使用并获取源代码，任何人都可以修改、发布甚至出售自己的应用。
- FreeSWITCH 支持 SIP、H323、Skype、Google Talk 等多种通信协议，并能很容易地与各种开源的 PBX 系统（如 sipXecs、Call Weaver、Bayonne、YATE 及 Asterisk 等）通信，它也可以与商用的交换系统（如华为、中兴的交换机或思科、Avaya 的交换机等）互通，如图 3-1 所示。

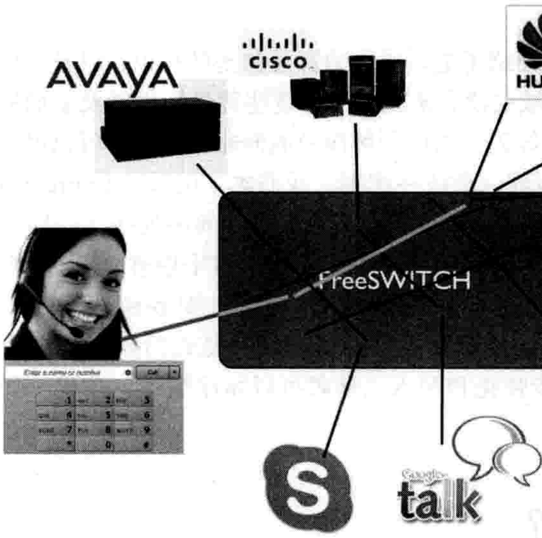


图 3-1　FreeSWITCH 支持多种协议

- FreeSWITCH 可以用作一个简单的交换引擎、一个 PBX、一个媒体网关或媒体支持 IVR 的服务器，或在运营商的 IMS 网络中担当 CSCF 或 Application Server 等。
- FreeSWITCH 遵循相关 RFC 并支持很多高级的 SIP 特性，如 Presence、BLF、SLA 以及 TCP、TLS 和 sRTP 等。它也可以用作一个 SBC 进行透明的 SIP 代理（proxy）以支持其他媒体，如 T.38 等。
- FreeSWITCH 支持宽带及窄带语音编码，电话会议桥可同时支持 8、12、16、24、32

[1] Raspberry Pi(http://www.raspberrypi.org/) 是一个装有 ARM CPU 的信用卡大小的计算机。事实上，笔者办公室的 IP-PBX 就是一台运行着 FreeSWITCH 的 Raspberry Pi。

[2] Mozilla Public License。详见：http://www.mozilla.org/MPL/1.1/。

及 48kHz 的语音。

- 从技术上讲，FreeSWITCH 是一个 B2BUA ⊖，它作为一个背靠背的用户代理用来帮助通信的双方进行实时的语音视频通信，如图 3-2 所示。



图 3-2　FreeSWITCH 是一个 B2BUA

3.1.2 FreeSWITCH 的功能

FreeSWITCH 是一个 B2BUA，所以它能做的工作非常多。在国外，很多 ISP 和运营商把它作为关键的软交换设备，处理成千上万路的并发通话；也有的把它用于呼叫中心，与各种企业级的应用系统（如 CRM、ERP 等）集成；在国内，也已经有很多应用案例，其被广泛用于金融、保险、电力、石油、煤炭等领域的呼叫中心、企业通信以及应急指挥调度平台等。从这一方面讲，它是传统的电话交换系统及商业的电话交换系统良好的替代品。除了简单的替代以外，它往往还提供更多的新功能、更灵活的数据集成能力和更快速的应用开发能力，在业务需求千变万化的今天显得格外有生命力。

另外，在当今的移动互联、物联网与大数据、云计算盛行的时代，好多厂商和互联网的创业者也把 FreeSWITCH 用于通信领域的“云”平台。FreeSWITCH 诞生的年代和背景、良好的设计架构以及活跃的技术支持社区都是它能在“云”平台上成功的坚实基础。

上面讲了 FreeSWITCH 的一些典型应用场景，下面看看它的典型功能：

- 在线计费、预付费功能
- 电话路由服务器
- 语音转码服务器
- 支持资源优先权和 QoS 的服务器
- 多点会议服务器
- IVR、语音通知服务器

⊖ Back-to-back User Agent，背靠背的用户代理。事实上，B2BUA 的概念会贯穿本书的始终，读者最好能理解它。

- VoiceMail 服务器
- PBX 应用和软交换
- 应用层网关
- 防火墙 /NAT 穿越应用
- 私有服务器
- 第三方呼叫控制应用
- 业务生成环境运行时引擎
- 会话边界控制器
- IMS 中的 S-CSCF/P-CSCF/I-CSCF
- SIP 网间互联网关
- SBC 及安全网关
- 传真服务器、T.30 到 T.38 网关

更多关于 FreeSWITCH 的特点和指标，可以参考 http://wiki.freeswitch.org/wiki/Specsheet。

3.2 快速体验

FreeSWITCH 的功能确实非常丰富和强大，在进一步学习之前我们先来一次完整的体验。

FreeSWITCH 默认的配置是一个 SOHO PBX（家用电话小交换机），那么我们本节的目标就是从零开始安装，实现分机互拨电话，测试各种功能，并通过添加一个 SIP-PSTN 网关拨打 PSTN 电话。这样，即使你没有任何使用经验，也应该能顺利学完本章，从而建立一个直观的认识。在体验过程中，你会遇到一点稍复杂的配置，如果不能完全理解，也不用担心，我们在后面会详细介绍。当然，如果你是一个很有经验的 FreeSWITCH 用户，那么大可跳过本章。

3.2.1 安装基本 FreeSWITCH 系统

在学习和使用 FreeSWITCH 之前，我们首先要安装一个基本的 FreeSWITCH 系统。FreeSWITCH 是跨平台的，大多数人使用各种 Linux 系统；很大一部分的开发者使用 Mac 平台进行开发；另外，也有很多用户在 Windows 平台上学习和使用它。因此，我们将分别介绍一下这几大主流平台的安装方法和应该注意的问题。FreeSWITCH 的开发非常活跃，因面版本更新很快，所以，我们首先从选择一个安装版本开始。

1. 版本简介

到本书截稿时止，FreeSWITCH 最新的版本是 1.4.beta。

FreeSWITCH 的版本号很有规律：版本号有 3 部分构成，以点隔开。其中，第 1 位为主版本号，第 2 位为次版本号，第 3 位用作补丁及更新的标志。其中，从第 2 位看，偶数的版本为稳定版，奇数的版本为开发版。开发版更新的内容在经过测试后会合并到稳定版中。如果有大的功能变化或改进，则稳定版和开发版版本两者的编号都会加 2。例如，上一个稳

定版本为 1.2，其对应的开发版为 1.3。最初的 1.2 由 1.2-rc1（Release Candidate，候选版）、1.2-rc2、到 1.2.0、1.2.1 等组成，到本书截稿时为止，最新的一个稳定版本是 1.2.22。

FreeSWITCH 使用 Git 进行版本控制。1.2 版本单独由一个 1.2.stable 的分支进行管理。其中，每一个发行版都会对应 Git 里的一个 Tag，如 v1.2.10、v1.2.12 等。而 1.2.stable 分支则永远是 1.2 版中最新的版本（可以看成是稳定分支中的不稳定版）。

FreeSWITCH 支持 32 位及 64 位的 Linux、Mac OS X、BSD、Solaris、Windows 等众多平台。某些平台上有编译好的安装包，但作者建议有一定基础的用户从源代码安装，因为这样便于版本的切换与升级。

在实际安装过程中，我们尽量选用比较新的版本。然而，某些版本在某些平台上有一些已知的问题，因此，具体的版本选择我们将在安装时再介绍。

2. 在 Windows 上安装

如果仅仅是为了学习和使用，在 Windows 平台上可以使用已经编译了的安装包。另外，为了完整性，本章也包含从源代码编译安装的步骤。本节假设读者已经熟悉 Windows 平台上的软件安装方法，在实际安装过程中仅对应该注意的事项加以说明。

（1）使用安装包安装

Windows 用户可以直接下载安装文件，下载地址为 http://files.freeswitch.org/windows/installer/。然后根据自己的系统选择不同目录，32 位系统的用户选择 x86 目录，64 位系统的用户选择 x64 目录。freeswitch.msi 是最新的安装程序，一般隔几天就会更新一次版本。笔者的测试环境是 32 位的 Windows XP，下载界面如图 3-3 所示。



图 3-3 下载 FreeSWITCH Windows 版

如同安装其他程序一样，我们全部选择默认设置即可，也就是说只要连续单击“Next”按钮就能安装完毕。安装完成后选择”开始菜单”→“所有程序”→“FreeSWITCH”→“FreeSWITCH”便可以启动 FreeSWITCH 了，启动后的界面如图 3-4 所示。

如果安装过程中你没有修改默认安装路径的话，那么 FreeSWITCH 的实际安装路径是：c:\Program Files\FreeSWITCH，配置文件在该目录的 conf 目录下。



图 3-4 Windows 上的 FreeSWITCH 控制台界面

（2）从源代码安装

如果从源代码安装，则首先要下载源代码。在此我们以 1.2.10 版为例，其下载地址是：http://files.freeswitch.org/freswitch-1.2.10.tar.gz。

除此之外，也可以 Git 仓库获取源代码。Git 是 FreeSWITCH 使用的版本控制工具，从 Git 仓库获取源代码的好处是可以随时更新，并可以很方便地切换到不同的代码分支，甚至“倒回”到任意提交点。

如果从 Git 仓库获取源代码，需要先在 Windows 上安装 Git。使用哪个 Git 版本不是很重要，笔者用的是从 https://code.google.com/p/msysgit/downloads/list?q=full+installer+official+git 下载的 1.8.3-preview 版。

安装 Git 很简单，一般来说双击安装文件并连续单击“Next”按钮即可安装完毕。不过，在 Windows 平台编译 FreeSWITCH 有几个要注意的事情，因此在安装 Git 的过程中我们也需要注意以下问题，并做适当的选择：

- 将 FreeSWITCH 的源代码放到一个“干净”的目录下。为避免有时候遇到奇怪的问题，最好把代码放到一个比较不容易出问题的目录下，如可以放到 C:\src\freeswitch 或 D:\src\freeswitch 下，这两个都是比较好的目录。而像 C:\My Documents（有空格）或 C:\ 源代码中文目录 \freeswitch（有中文）之类的则在编译或使用时可能会有问题。
- Git 相关的环境变量。Git 是从 UNIX 系统上移植过来的一个命令行工具，因此需要一些相关的环境变量。在安装时有三个选项（见图 3-5），笔者建议使用第三项，这样最省心。当然，第三项与 Windows 系统的命令会有少量冲突，如 find 等。但实际上，你可能永远不会用到 Windows 上的命令行工具，因此，在安装过程中果断选择第三项可以省去不少麻烦。

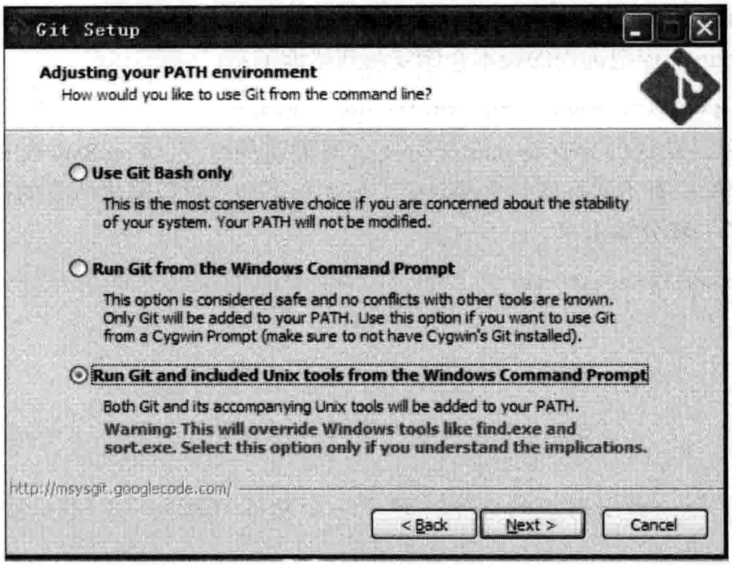


图 3-5 安装 Git 时选择自动包含所有相关的环境变量

- 关闭 Git 的自动换行符转换。众所周知，Widows 使用“回车 + 换行”（“\r\n”，又称作“CRLF”）做换行符，而 UNIX 仅使用“\n”。Git 可以自动在不同的换行符间转换。但问题是，有时候自动转换不靠谱，尤其是对于 FreeSWITCH 这样大型的项目，所以笔者一般在安装 Git 时就关掉这一选项（否则在编译阶段可能会出奇怪的错误），如图 3-6 所示。

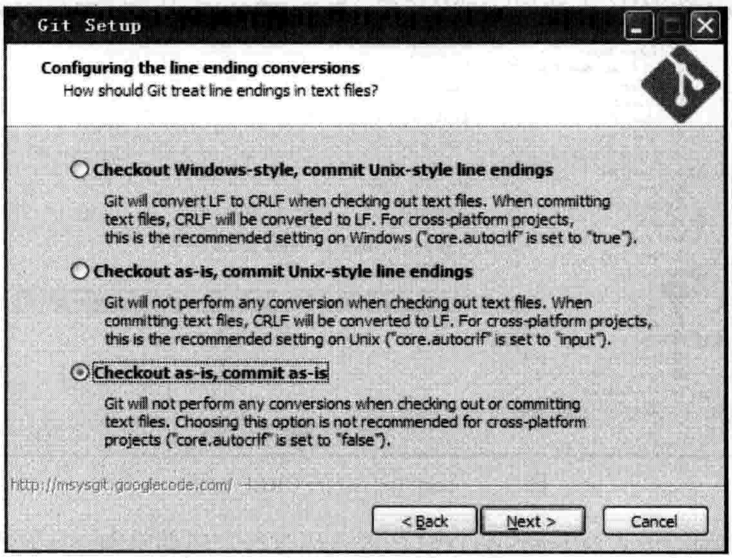


图 3-6 不使用自动换行符转换功能

接下来可以连续按“ Next”按钮直到安装完毕。Git 安装完毕后就可以切换到命令行方式，使用 git clone 命令把远程的版本仓库复制到本地了：

```
git clone git@git.freeswitch.org/freeswitch.git
```

复制完毕后，默认的分支是 master 分支，即最新的分支。FreeSWITCH 对不同版本的安装包在 Git 仓库中有不同标签与之相对应。使用如下命令可以列出所有的标签（tag，为节省篇幅，省略了一部分输出）：

```
C:\src\freeswitch> git tag
v1.2.0
v1.2.1
v1.2.10
v1.2.21
v1.2.22
v1.2.9
v1.5.7
```

可以用以下命令检出对应的标签并建立一个新的本地分支，（我们在这里仍然使用 1.2.10 版）：

```
C:\src\freeswitch> git checkout -b v1.2.10
Switched to a new branch 'v1.2.10'
```

当然，如果你不习惯使用这种命令和工具，则可以下载 Tortoise Git 图形界面工具，下载地址为 https://code.google.com/p/tortoisegit/wiki/Download。

Tortoise Git 也允许通过 AutoCrlf 复选框选择是否开启自动换行符转换，为避免它自动转换，我们应该保证该复选框是非选中状态的，如图 3-7 所示。

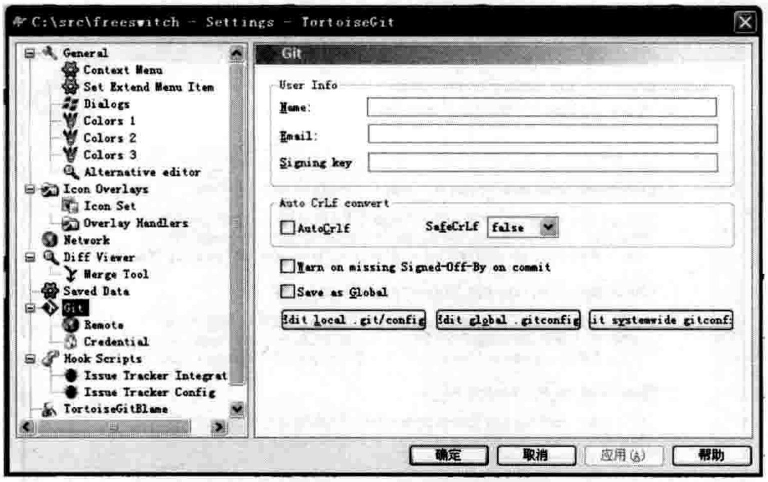


图 3-7 Tortoise Git 的 CRLF 设置

使用图形界面的方式对 FreeSWITCH 的源代码进行复制会比命令行方式直观一些，如图 3-8 所示。

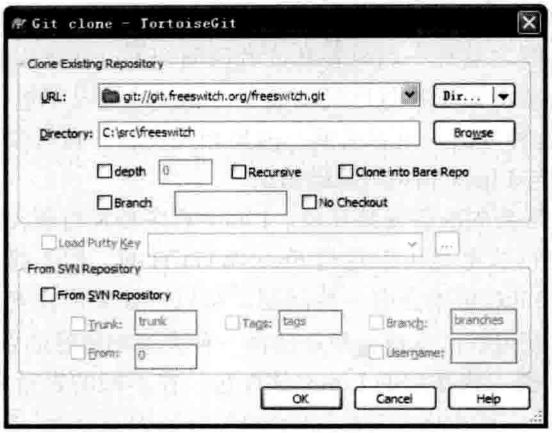


图 3-8　使用 Tortoise Git 复制 FreeSWITCH 代码库

复制完毕后，可以使用右键菜单，通过选择相应的菜单项检出（checkout）相应的标签或分支，在这里就不多介绍了。

有了 FreeSWITCH 源代码，接下来还需要下载编译工具。Microsoft 提供 Visual Studio 工具进行开发。FreeSWITCH 中有 VS2005、VS2008、VS2010 以及 VS2012 的工程文件。VS2008 及以前的支持已经不再更新了，因此不推荐使用。VS2010 和 VS2012 目前是官方支持的版本。在此，笔者使用 VS2010 Express 版为例加以说明。

FreeSWITCH 的源代码目录下有一个名为 Freeswitch.express.2010.sln 的 Solution 文件，双击鼠标打开它，然后选择菜单项“调试”→“生成解决方案”，或按快捷键 F7，就可以进行编译了。不出问题的话，编译成功后将会在源代码目录下的 Win32 目录下出现 Debug 或 Release 目录（取决于编译前的选择，默认为 Debug），编译完成的目标文件都会在这些目录下。

图 3-9 所示是使用 VS2010 正在编译 FreeSWITCH 源代码时的界面。



图 3-9　在 Windows 上使用 VS2010 编译

3. 在 Linux 系统上安装

在开始本小节的讲解之前有一点需要和读者声明一下，就是以下内容是在假定读者已经有了一定的 Linux 的基本知识并且已经安装了 Linux 的情况下进行的。若读者没接触过 Linux，则建议不采用这种方法，或者去网上搜集相关资料，自行学习 Linux 相关知识。限于篇幅，本书不再介绍与 Linux 相关的基础知识。

在安装之前，我们需要先准备安装环境。Linux 有多种发行版（发行套件）。一般来说，大部分主流的 Linux 发行版都是可以运行 FreeSWITCH 的，但不排除某些发行版的内核、文件系统、编译环境、LibC 版本会有一些问题。所以，如果你在安装或使用 FreeSWITCH 的过程中遇到问题时想获得社区支持，最好选择一种大家都熟悉的发行套件[⊖]。另外，编译安装 FreeSWITCH 要依赖一些基础的 Linux 软件包，在不同的发行版平台上可以用以下不同的命令安装：

CentOS：

```
yum install -y autoconf automake libtool gcc-c++ ncurses-devel make zlib-devel libjpeg-devel
yum install -y openssl-devel e2fsprogs-devel curl-devel pcre-devel speex-devel sqlite-devel
```

Ubuntu/Debian:

```
apt-get -y install build-essential automake autoconf git-core wget libtool
apt-get -y install libncurses5-dev libtiff-dev libjpeg-dev zlib1g-dev libssl-dev libsqlite3-dev
apt-get -y install libpcre3-dev libspeexdsp-dev libspeex-dev libcurl4-openssl-dev libopus-dev
```

除此之外，如果你想从 Git 仓库中下载源代码安装 FreeSWITCH，则需要事先安装 Git。CentOS 5 默认的软件仓库中可能没有 Git，如果你需要在 CentOS 5 上使用 Git 安装，则可以先安装 rpmforge（http://pkgs.repoforge.org/rpmforge-release/），然后再安装 Git。CentOS 6 的 yum 源中已经包含了 Git，因此不需要 rpmforge 了。关于如何在你的发行版上安装 Git 请参考有关资料。一般来说，在 Ubuntu 或 Debian 上可以使用如下命令来安装：

```
apt-get install git-core
```

在 CentOS 6 上则可以使用如下命令：

```
yum install git
```

在准备好相关 Linux 环境以后，就可以安装 FreeSWITCH 了。以下的安装步骤跟选用哪种 Linux 发生套件关系不大。从以下三种安装方式可任选其一，默认安装位置都是 /usr/

⊖ FreeSWITCH 开发者使用的平台是 Debian 7，社区中也有许多人在使用 CentOS（5、6）和 Ubuntu。在将要发布的 FreeSWITCH1.4 版中，官方已决定不再支持 Debian6 及 CentOS5 等低版本的 Linux 系统，但 FreeSWITCH1.2 版仍支持，总之选择操作系统的原则是，如果你对 FreeSWITCH 和 Linux 都不熟悉的话，建议选择最新版本的 Debian 或 CentOS，这样遇到问题可以比较方便地得到帮助（开发者并不使用 Ubuntu）。如果你选择一个生僻的或比较小众的发行版，若遇到问题则很难找到能帮助你的人。当然，如果你是往新的系统移植，建议至少熟悉了 FreeSWITCH 以后再做。

local/freeswitch。安装过程中会下载源代码目录，请保留，以便以后升级及安装配置其他组件。

（1）从 Git 仓库安装

从代码库安装能让你永远使用最新的版本，如果安装过程中遇到问题也能够方便地回退到先前的版本。首先我们使用下列命令来从 Git 仓库中获取 FreeSWITCH 的源代码：

```
git clone git://git.freeswitch.org/freeswitch.git
```

如果需要安装特定的版本，则可以切换到对应的 Tag。如安装 1.2.22，你可以执行：

```
cd freeswitch                        # 进入源代码目录
git checkout -b v1.2.12              # 根据一个 Tag 检出到一个本地分支
```

或

```
git checkout -b v1.4.beta            # 从远程分支检出一个本地分支
```

当然，如果对 Git 比较熟悉，你也可以直接在复制时指定一个分支：

```
git clone -b v1.4.beta git://git.freeswitch.org/freeswitch.git
```

总之，在 Linux 上得到源代码并检出适当的 Tag 或分支（新手推荐选择安装时最新的稳定版）后，便可以执行下列命令进行安装（注意下列命令要在 FreeSWITCH 源代码目录中执行）：

```
./bootstrap.sh
./configure
make install
```

上面的命令是在 Linux 上从源代码安装软件的标准过程。首先第 1 行执行 bootstrap.sh 以初始化一些编译环境，第 2 行配置编译环境，第 3 执行编译安装。

（2）解压缩源码包安装

注意，这里我们使用本书截稿时最新的 1.4.beta6 版，如果你安装的时候，应该检查一下是否有更新的版本出现。

使用 wget 可以获取源代码安装包。下列命令会首先使用 wget 下载安装包，然后使用 tar 解压缩，最后使用 cd 命令进入源代码目录：

```
wget http://files.freeswitch.org/freeswitch-1.4.0.beta6.tar.bz2
tar xvjf freeswitch-1.4.0.beta6.tar.bz2
cd freeswitch-1.4.0
```

接下来的配置安装就很简单了，具体如下：

```
./configure
make install
```

可以看到，与上一种方法不同的是，它不需要执行 bootstrap.sh（源代码在打成 tar 包前已经执行过了，因而不需要 automake 和 autoconf 工具），便可以直接配置安装。

（3）最快安装

这是史上最快的安装方式，如果你对 UNIX 类的编译系统比较熟，或者跟作者一样需要

经常安装系统，你不妨试一试这种方式[㊀]：

```
wget http://www.freeswitch.org.cn/Makefile && make install
```

以上命令会使用 wget 下载一个 Makefile，然后使用 make 执行安装过程。安装过程中它会从 Git 仓库中获取代码[㊁]，实际上执行的操作跟前面的安装方式相同。

4. 在 Mac 系统上安装

苹果公司的 Mac 系统是理想的开发者平台，尤其是苹果 iPhone 和 iPad 在全世界范围内的成功，使得该平台吸引了大量的开发者。而且大多数的 FreeSWITCH 开发者也都在使用 Mac。事实上，本书就是在 Mac 系统上使用 Sublime Text 2 编辑器写成的，本书的大部分环境和截图也是在 Mac 系统上做的。

如果你想在 Mac[㊂]系统上安装 FreeSWITCH，则需要先下载安装 Apple 的 Xcode 工具（https://developer.apple.com/xcode/），并选择菜单 Preferences -> Downloads 安装命令行工具（Command Line Tools）[㊃]，如图 3-10 所示。

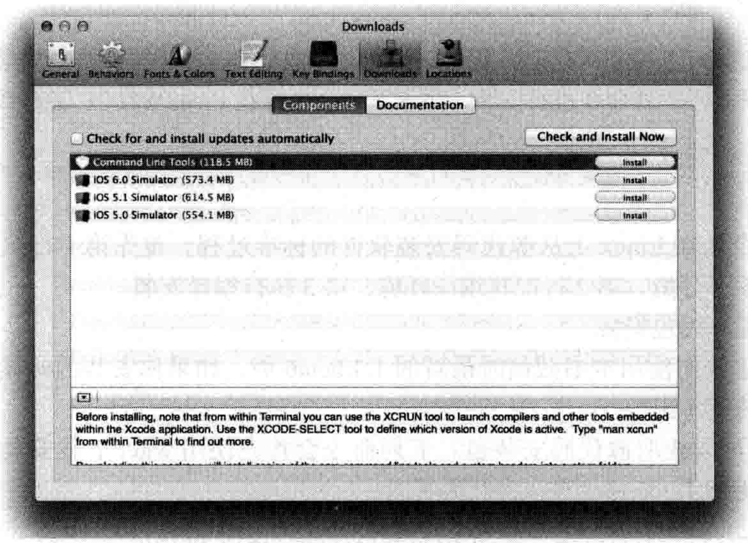


图 3-10 安装 Apple Xcode 及 Command Line Tools

除此之外，FreeSWITCH 也依赖于一些第三方的库。要安装第三方的库，在 Mac 平台

㊀ 在本书截稿时，FreeSWITCH 1.4 还没有发布正式版，因而你拿到的最新的 FreeSWITCH 版本在实际安装时可能跟上面描述的安装过程有出入。如果按照上面几种安装方式遇到问题的话，尝试读一下这里提到的 Makefile 的文件的内容也许会有帮助。

㊁ 根据发行版的不同它还可以自动选择使用 apt-get（Debian/Ubuntu）、yum（CentOs）或 Homebrew（MacOSX）来安装依赖的包和库。对此感兴趣的读者可以自行看一下下载得到的 Makefile 文件。

㊂ 本例是在 Mac OS X 10.8.4 上做的，Apple 公司在 2013 年 10 月份发布了 Mac OS X 10.9，但笔者尚没有升级。

㊃ 本例是在 Mac10.84 上写的，如果在 Mac10.9 上，请使用 xcode-select-install 命令安装。

上一般使用 Macports、Flink 和 Homebrew 等包管理工具。Homebrew 是比较新的工具，安装和使用起来都很方便。如果你还没有安装，可以用以下命令安装[1]：

```
ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
```

安装完 Homebrew 后，可以先试一下安装 Git 和 libtiff 库，安装 FreeSWITCH 时也需要用到它们：

```
brew install git
brew install libtiff
```

在你的系统上使用以上命令时，若系统提示你没有权限，则可以在命令前面加上 sudo，如安装 libtiff 库的命令就可写为：

```
sudo brew install libtiff
```

其他的安装步骤就全部跟 Linux 上一样了，如：

```
git clone git://git.freeswitch.git/freeswitch.git
cd freeswitch
./bootstrap.sh
./configure
make install                    # 或 sudo make install（根据是否有权限）
```

5. 安装声音文件

在后面的例子中我们需要一些声音文件。声音文件有两种，一种是提示音，用于通话期间的语音提示，如 VoiceMail 的提示音，支持 TTS 功能的提示音等；另一种是音乐，用于在 Hold 状态时播放，即所谓的 Music on Hold（MOH）[2]。

在 Windows 系统上，这些声音文件是默认安装的。而在 Linux 或 Mac 上安装这些声音文件也异常简单。你只需在源代码目录中执行：

```
make sounds-install
make moh-install
```

安装过程中将自动从 files.freeswitch.org 下载相关的语音包，并解压缩到相关的安装路径中（默认安装在 /usr/local/freeswitch/sounds 下）。

另外，FreeSWITCH 支持 8kHz、16kHz、32kHz 及 48kHz 的语音[3]。与上面的声音文件相对应的高清声音文件可以选择安装。如以下命令安装 16kHz 的声音文件：

```
make cd-sounds-install
make cd-moh-install
```

---

[1] 由于 Homebrew 的安装地址可能变化，请到官方网站（http://mxcl.github.io/homebrew/index_zh-cn.html）查看最新的安装方法。

[2] 在实际应用中，如果通话的双方中有一方不想让对方听到自己的讲话，或者在不挂机的情况下拨打另一路电话时，就需要将电话置于 Hold（保持）状态，这时候 FreeSWITCH 需要向对方播放保持音乐，即 MOH。另外，在拨打某些呼叫中心客户号码的时候经常遇到"座席繁忙，请等待…"，然后是一连串的音乐，这些音乐也可以称为 MOH。

[3] 很少有其他电话系统支持如此多的抽样频率。普通电话是 8kHz，某些新的 IP 话机支持更高的抽样频率，更高频率意味着更好的语音质量（细节听起来更逼真）。

6. 安装完成后的操作

FreeSWITCH 使用 make install 安装完成后，会显示一个有用的帮助，它会提示你接下来可以用哪些 make 命令执行一些其他的操作（如我们刚才安装声音文件的命令，在这里就可以看到）。下面笔者在默认的帮助信息后增加了一些中文的注释，读者可以在学习中自行练习一下。

```
+---------- FreeSWITCH install Complete ----------+
+ FreeSWITCH has been successfully installed.     +
+                                                 +
+       Install sounds:                           +  安装声音文件
+       (uhd-sounds includes hd-sounds, sounds)   +
+       (hd-sounds includes sounds)               +
+       ------------------------------------      +
+               make cd-sounds-install            +  CD 音质的声音文件
+               make cd-moh-install               +
+                                                 +
+               make uhd-sounds-install           +  超高清声音文件
+               make uhd-moh-install              +
+                                                 +
+               make hd-sounds-install            +  高清声音文件
+               make hd-moh-install               +
+                                                 +
+               make sounds-install               +  标准声音文件
+               make moh-install                  +
+                                                 +
+       Install non english sounds:               +  安装其他语言的声音文件
+       replace XX with language                  +
+       (ru : Russian)                            +  如 ru 代表俄语
+       ------------------------------------      +
+               make cd-sounds-XX-install         +
+               make uhd-sounds-XX-install        +
+               make hd-sounds-XX-install         +
+               make sounds-XX-install            +
+                                                 +
+       Upgrade to latest:                        +  升级到最新版本
+       ----------------------------------        +
+               make current                      +
+                                                 +
+       Rebuild all:                              +  重新编译
+       ----------------------------------        +
+               make sure                         +
+                                                 +
+       Install/Re-install default config:          +  安装（或重新安装）配置文件
+       ----------------------------------        +
+               make samples                      +
+                                                 +
+       Additional resources:                     +
+       ----------------------------------        +
+       http://www.freeswitch.org                 +  官方网站
+       http://wiki.freeswitch.org                +  官方 Wiki
+       http://jira.freeswitch.org                +  官方的缺陷跟踪工具
+       http://lists.freeswitch.org               +  邮件列表
+                                                 +
```

```
+        irc.freenode.net / #freeswitch           +  IRC 聊天室
+-------------------------------------------------+
```

至此，FreeSWITCH 就已经安装完了。在 UNIX 类操作系统上，其默认的安装位置是 /usr/local/freeswitch（下文所述的路径全部相对于该路径）。两个常用的命令是 bin/freeswitch 和 bin/fs_cli（我们下面会讲到它们的用法），为了便于使用，建议将这两个命令做符号链接放到你的搜索路径中，如：

```
ln -sf /usr/local/freeswitch/bin/freeswitch /usr/bin/
ln -sf /usr/local/freeswitch/bin/fs_cli /usr/bin/
```

接下来 FreeSWITCH 就应该可以启动了。通过在终端中执行 freeswitch 命令 (如果你已做符号链接的话，否则要执行 /usr/local/freeswitch/bin/freeswitch) 可以将 FreeSWITCH 启动到前台。启动过程中会有许多 log 输出，第一次启动时会有一些错误和警告，可以不必理会⊖。启动完成后会进入系统控制台，并显示类似的提示符 " freeswitch@localhost>" (以下简称 freeswitch>)。通过在控制台中输入 shutdown 命令可以关闭 FreeSWITCH。

如果您想将 FreeSWITCH 启动到后台 (Daemon，服务模式)，可以使用 freeswitch -nc (即 No console)。后台模式没有控制台，如果想关闭 FreeSWITCH，可以直接在 Linux 提示符下通过 freeswitch -stop 命令实现。

不管 FreeSWITCH 运行在前面还是后台，都可以使用客户端软件 fs_cli 连接到它并对它进行控制。使用方法为：

```
/usr/local/freeswwitch/bin/fs_cli
```

当然，如果上面已经做了符号连接也可以直接运行 fs_cli。任何时间想退出 fs_cli 客户端，都可以输入 /exit 或按 Ctrl + D 组合键，也可以直接关掉终端窗口。

3.2.2 连接 SIP 电话

FreeSWITCH 最典型的应用是作为一个服务器（它实际上是一个背靠背的用户代理，即 B2BUA），并用电话客户端软件（一般叫软电话）连接到它。虽然 FreeSWITCH 支持 IAX、H323、Skype、Gtalk 等众多通信协议，但其最主要的协议还是 SIP。支持 SIP 的软电话有很多，笔者比较常用的是 X-Lite 和 Zoiper。这两款软电话都支持 Linux、Mac OS X 和 Windows 平台，免费使用但是不开源。在 Linux 上你还可以使用 Ekiga 软电话，它是开源的。

强烈建议在同一局域网上的其他机器上安装软电话，并确保麦克风和耳机可以正常工作。当然，如果你没有多余的机器做这个实验，也可以在同一台机器上安装。只是需要注意，软电话不要占用 UDP 5060 端口，因为 FreeSWITCH 默认要使用该端口，这是新手常会遇到的一个问题。你可以通过先启动 FreeSWITCH 再启动软电话来避免该问题（后者如果在启动时发现 5060 端口已被占用，一般会尝试选择其他端口），另外有些软电话允许你修改本

⊖ FreeSWITCH 在第一次启动时由于没有必要的数据库表，它会打印一些出错信息，并自创建这些表。只要下次启动时不出现错误，就可以暂时不必理会。

地监听端口⊖。

在 UNIX 类平台上，通过输入以下命令可以知道 FreeSWITCH 监听在哪个 IP 地址上，记住这个 IP 地址（:5060 以前的部分），下面要用到：

```
netstat -an | grep 5060

udp        0       0 192.168.0.9:5060          0.0.0.0:*
```

FreeSWITCH 默认配置了 1000 ~ 1019 共 20 个用户，你可以随便选择一个用户进行配置，配置过程如下：

1）在 X-Lite 上右击，选 “Sip Account Settings...”，单击 “Add” 添加一个账号，填入以下参数（Zoiper 可参照配置）：

```
Display Name: 1000
User name: 1000
Password: 1234
Authorization user name: 1000
Domain: 你的 IP 地址，就是刚才你记住的那个
```

2）其他都使用默认设置，单击 “OK” 按钮就可以了。然后单击 “Close” 按钮关闭 Sip Account 设置窗口。这时 X-Lite 将自动向 FreeSWITCH 注册。注册成功后会显示 “Ready. Your username is 1000”，另外，左侧的 “拨打电话”（Dial）按钮会变成绿色的，如图 3-11 所示。

值得一提的是，笔者使用的是一个旧版本的 X-Lite，之所以这么做，是因为考虑到大家可能对这个版本的 X-Lite 更熟悉一些。新版本的 X-Lite 界面如图 3-12 所示。

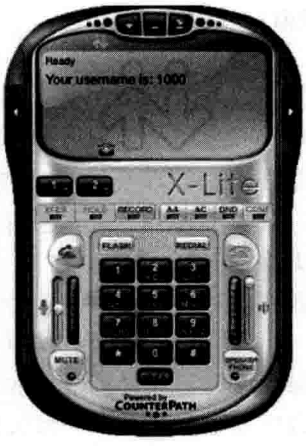


图 3-11 XLite 软电话注册后

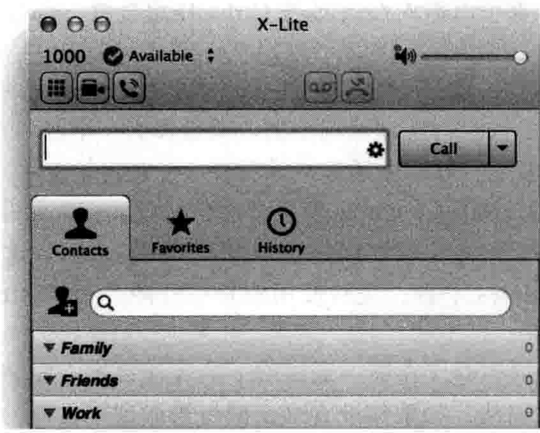


图 3-12 XLite 软电话（新版）

激动人心的时刻就要来了。输入 “9664” 按回车（或按绿色拨打电话按钮），就应该能

⊖ 特别注意，如果你是在 Linux 上，并且在同一台机器上使用 Ekiga 的话，肯定会遇到这个问题。你需要手工使用 gconf_editor 来更改 Ekiga 使用的端口，当然，也可以改 FreeSWITCH 的端口，如果你会的话。

听到保持音乐（MOH）。如果听不到也不要气馁，看一下控制台上有没有提示什么错误。如果有“File Not Found”之类的提示，多半是声音文件没有安装，重新查看 make moh-install 是否有错误。接下来，可以依次试试拨打表 3-1 所示的号码。

表 3-1 默认号码及说明

号码	说明
9664	保持音乐
9191	注册 ClueCon
9192	在 log 中显示 Channel 信息
9195	echo，回音测试，延迟 5 秒
9196	echo，回音测试
9197	milliwatte extension，铃音生成
9198	TGML 铃音生成示例
9180	铃音测试，使用远端生成的回铃音
9181	铃音测试，产生英式铃音
9182	铃音测试，使用音乐当铃音，彩铃
9183	先应答，然后发送英式铃音
9184	先应答，然后发送音乐铃音
9178	收传真
9179	发传真
5000	示例 IVR
4000	听取语音信箱
33xx	电话会议，48kHz（其中 xx 可为 00 ~ 99，下同）
32xx	电话会议，32kHz
31xx	电话会议，16kHz
30xx	电话会议，8kHz
2000-2002	呼叫组
1000-1019	默认分机号

详情见 http://wiki.freeswitch.org/wiki/Default_Dialplan_QRF。

另外，也许你想尝试注册另外一个 SIP 用户并在两者间通话。此时最好是在同一个局域网中的另外一台机器上启动另一个 X-Lite ，并使用 1001 注册，注册完毕后就可以在 1000 上呼叫 1001，或在 1001 上呼叫 1000 。当然，你仍然可以在同一台机器上做这件事（比方说用 Zoiper 注册为 1001），需要注意的是，由于你机器上只有一个声卡，两者可能会争用声音设备。特别是在 Linux 上，有些软件会独占声音设备。如果同时也有一个 USB 接口的耳机，那就可以设置不同的软件使用不同的声音设备。

如果你手边有硬件的 IP 话机，你也可以试一试。与传统的话机相比，IP 话机更加“智能”，功能也更丰富。因为硬件话机的设置方法和软件电话大同小异，所以只要明白上述软电话的设置，即可知道如何设置硬件话机了。我国产的话机质优价廉，在国际上都有

很好的口碑。下面我们分别以国产的亿联和潮流的话机为例，熟悉一下硬件话机注册到 FreeSWITCH 的配置。

亿联（Yealink[1]）话机是在国内能找到的质量比较好的话机，而且它有好多独有的特性。我们在后面的章节会讲到它的其他特性，这里我们先看看基本的配置。话机本身有一个液晶显示屏，并可以通过按键设置账号信息，但那样配置起来比较烦琐。在液晶屏上找到话机的 IP 地址以后[2]，用浏览器打开，界面如图 3-13 所示。



图 3-13 亿联话机的账号配置界面

账号配置跟软电话差不多，“显示名称”可以随便填，“注册名称”和“用户名称”这里我们都用 1002，“密码”是默认的 1234，“SIP 服务器”处输入你的 IP 地址，其他的都保留默认设置，然后单击“提交”按钮。如果一切顺利，就能看到“账号状态”显示为“注册上”，这时就可以拨打 1000 或 1001 了。

潮流（Grandstream[3]）话机也是质量不错的话机，配置和使用也比较方便。它的配置界面如图 3-14 所示。其中“账号名”可以随便填，“SIP 服务器”中输入你的 IP 地址，“SIP 用户 ID”、“认证 ID”及“名称”都填入 1003，“密码”也是默认的 1234。保存并提交后即可注册。潮流话机的注册状态是在单独的“状态”页面中显示的。

笔者使用这几款话机注册后相互拨打，彼此都能通，声音质量也很不错。

---

[1] www.yealink.cn。

[2] 一般使用 DHCP 启动后会自动获得一个 IP 地址，否则，也可以设一个静态的 IP。

[3] www.grandstream.com.cn。



图 3-14　潮流话机的账号配置界面

3.3　配置 FreeSWITCH

FreeSWITCH 配置文件默认放在 conf/ 下，它由一系列 XML 配置文件组成。最顶层的文件是 freeswitch.xml，系统启动时它依次装入其他一些 XML 文件并最终组成一个大的 XML 文件。基本的目录结构和主要配置文件如表 3-2 所示。

表 3-2　配置文件的目录结构

conf/ 目录和文件	说明
├── vars.xml	一些常用变量
├── switch.xml	主配置文件，它会使用 include 语句装入其他文件
├── autoload_configs	目录，存放自动加载的配置文件
│　├── modules.conf.xml	配置当 FreeSWITCH 启动时自动装载哪些模块
│　├── *.xml	一般来说每个模块都有一个配置文件
├── chatplan	聊天计划
├── dialplan	拨号计划
│　├── default.xml	默认的拨号计划配置，一般用于内部用户路由
│　├── public.xml	默认的拨号计划配置，一般用于外部来话路由
├── directory	用户目录
│　└── default	默认的用户目录配置
│　　　├── *.xml	SIP 用户，每用户一个文件

（续）

conf/ 目录和文件	说明
├── ivr_menus	IVR 菜单
├── jingle_profiles	连接 Google Talk 的相关配置
├── lang	多语言支持
│ ├── en	英语
│ └── fr	法语
├── mrcp_profiles	MRCP 的相关配置，用于跟第三方语音合成和语音识别系统对接
├── sip_profiles	SIP 配置文件
│ ├── internal.xml	一个 SIP profile，或称作一个 SIP-UA，监听在本地 IP 及端口 5060
│ │	一般供内网用户使用
│ ├── externa.xml	另一个 SIP-UA，用作外部连接，端口 5080
└── skinny_profiles	思科 SCCP 协议话机的配置文件

下面我们先通过学习添加一个新的 FreeSWITCH 用户来简单熟悉一下 FreeSWITCH 的配置文件。

FreeSWITCH 默认设置了 20 个用户（1000 ~ 1019），如果你需要更多的用户，或者想通过添加一个用户来学习 FreeSWITCH 配置，只需要简单执行以下三步：

1）在 conf/directory/default/ 中增加一个用户配置文件。

2）修改拨号计划（Dialplan）使其他用户可以呼叫到它。

3）重新加载配置使其生效。

例如我们想添加用户 Jack，分机号是 1234。只需要到 conf/directory/default 目录下，将 1000.xml 复制到 1234.xml 中。打开 1234.xml，将所有 1000 都改为 1234。并把 effective_caller_id_name 的值改为 Jack，然后存盘退出，命令如下：

```
<variable name="effective_caller_id_name" value="Jack"/>
```

接下来，打开 conf/dialplan/default.xml，找到下面一行

```
<condition field="destination_number" expression="^(10[01][0-9])$">
```

将其改为

```
<condition field="destination_number" expression="^(10[01][0-9]|1234)$">
```

熟悉正则表达式的读者应该知道，“^(10[01][0-9])$”匹配被叫号码 1000 ~ 1019。因此我们修改之后的表达式就多匹配了一个 1234。FreeSWITCH 使用 Perl 兼容的正则表达式（PCRE）。

现在，回到控制台或启动 fs_cli，执行 reloadxml 命令或按快捷 F6，使新的配置生效。

找到刚才注册为 1001 的软电话（或启动一个新的，如果你有足够的机器的话），把 1001 都改为 1234 然后重新注册，这时就可以与 1000 相互进行拨打测试了。如果没有多台机器，在同一台机器上运行多个软电话可能有冲突，这时可以直接进入 FreeSWITCH 控制台使用如下命令进行测试：

```
freeswitch> sofia status profile internal reg      （显示多少用户已注册）
freeswitch> originate user/1000 &echo             （同上）
freeswitch> originate user/1000 9999              （相当于在软电话 1000 上拨打 9999）
freeswitch> originate user/1000 9999 XML default （同上）
```

其中，echo 程序是一个很简单的程序（App），它只是将你说话的内容原样再放给你听，在测试时很有用，在本书中我们会经常用它来测试。

3.4 FreeSWITCH 用作软电话

也可以把 FreeSWITCH 简单地用作一个软电话（可以看作用 FreeSWITCH 做了一个 X-Lite）。虽然相比而言，FreeSWITCH 比配置 X-Lite 略微复杂一些，但你会从中得到更多好处：FreeSWITCH 是开源的，更强大、灵活。关键是它是目前笔者所知道的唯一支持 CELT 高清通话的软电话。

FreeSWITCH 使用 mod_portaudio 模块支持你本地的音频设备，该模块默认是不编译的。在你的源代码目录下执行如下命令，以安装该模块：

```
make mod_portaudio
make mod_portaudio-install
```

其他的模块也可以依照上面的方式进行重新编译和安装。安装完成后到控制台中执行：

```
freeswitch> load mod_portaudio
```

如果得到“Cannot find an input device”之类的错误，则可能是你的声卡驱动有问题。如果是提示“+OK”就是成功了。接着执行 pa devlist 命令，可以看到如下输出：

```
freeswitch> pa devlist

API CALL [pa(devlist)] output:
0;Built-in Microphone;2;0;
1;Built-in Speaker;0;2;r
2;Built-in Headphone;0;2;
3;Logitech USB Headset;0;2;o
4;Logitech USB Headset;1;0;i
```

以上是笔者的笔记本电脑上的输出，它列出了所有的声音设备。其中，3 和 4 最后的“o”和“i”分别代表声音输出（out）和输入（in）设备。在你的电脑上可能不一样，如果你想选择其他设备，可以使用命令进行修改。例如下列命令可以选择使用笔者电脑上内置的麦克风和耳机：

```
freeswitch> pa indev #0
freeswitch> pa outdev #2
```

至此你就有了一个可以用命令行控制的软电话了。尝试输入以下命令：

```
freeswitch> pa looptest      （回路测试，echo）
freeswitch> pa call 9196     （呼叫 9196）
freeswitch> pa call 1000     （呼叫 1000）
freeswitch> pa hangup        （挂机）
```

如上所示，你可以呼叫刚才试过的所有号码。现在假设想从 SIP 分机 1000 呼叫到你，那么需要修改拨号计划 (Dialplan)。用你喜欢的编辑器编辑以下文件并放到 conf/dialplan/default/portaudio.xml 中：

```
<include>
  <extension name="call me">
    <condition field="destination_number" expression="^(me|12345678)$">
      <action application="bridge" data="portaudio"/>
    </condition>
  </extension>
</include>
```

然后，在控制台中按 F6 或输入以下命令使之生效：

```
freeswitch> reloadxml
```

在分机 1000 上呼叫 me 或 12345678（你肯定想为自己选择一个更酷的号码），然后在控制台上应该能看到类似 [DEBUG] mod_portaudio.c:268 BRRRRING! BRRRRING! call 1 的输出（如果看不到，按 F8 能得到详细的 log），这说明你的软电话在振铃。多按几个回车，然后输入 pa answer 就可以接听电话了。输入 pa hangup 可以挂断电话。

当然，你肯定希望在振铃时能听到真正的振铃音而不是看什么 BRRRRRING。好办，选择一个好听的声音文件（WAV 格式），编辑 conf/autoload_configs/portaudio.conf.xml，将 ring-file 一行修改为下面的样子，其中，value 指定你的声音文件的路径：

```
<param name="ring-file" value="/home/your_name/your_ring_file.wav"/>
```

然后重新加载模块：

```
freeswitch> reloadxml
freeswitch> reload mod_portaudio
```

再打打试试，看是否能听到振铃音了？

如果你用不习惯字符界面，可以看一下 FreeSWITCH-Air（http://www.freeswitch.org.cn/download），它是使用 Adobe Air 开发的，为 FreeSWITCH 提供一个简洁的软电话的图形界面。另外，如果你需要高清通话，除需要设置相关的语音编解码器（codec）外，你还需要有一个好的耳机才能达到最好的效果。笔者使用的是一款 Logitech 的 USB 耳机。除此之外，还有两款基于 FreeSWITCH 的软电话，分别是 FSComm[㊀]（QT 实现）和 FSClient[㊁]（C# 实现）。

3.5 配置 SIP 网关拨打外部电话

如果你拥有某个运营商提供的 SIP 账号，那么你就可以通过配置 SIP 来拨打外部电话了。该 SIP 账号（或提供该账号的设备）在 FreeSWITCH 中称为 SIP 网关（Gateway）。添加一个网关只需要在 conf/sip_profiles/external/ 中创建一个 XML 文件，名字可以随便起，如

㊀ http://wiki.freeswitch.org/wiki/FSComm

㊁ http://wiki.freeswitch.org/wiki/FSClient

gw1.xml，然后在该文件中输入如下代码：

```
<gateway name="gw1">
    <param name="realm" value="SIP 服务器地址，可以是 IP 或 IP: 端口号 "/>
    <param name="username" value="SIP 用户名 "/>
    <param name="password" value=" 密码 "/>
</gateway>
```

如果你的 SIP 网关还需要其他参数，可以参阅同目录下的 example.xml，但一般来说上述参数就够了⊖。你可以重启 FreeSWITCH，或者执行以下命令使用之生效：

```
freeswitch> sofia profile external rescan
```

显示一下网关的注册状态：

```
freeswitch> sofia status
```

如果显示 gateway gw1 的状态是 REGED，则表明已正确地注册到了网关上。你可以先用命令试一下网关是否工作正常：

```
freeswitch> originate sofia/gateway/gw1/xxxxxx &echo
```

以上命令会通过网关 gw1 呼叫号码 xxxxxx（可能是你的手机号），被叫号码接听电话后，FreeSWITCH 会执行 echo 程序，你应该就能听到自己的回音了。

当然，世界 SIP 网关五花八门，你实际操作起来可能不会这么顺利。如果真的在这里遇到问题，那么你大可继续往下阅读，相信你读完本书之后，跟任何网关对接的复杂问题都能迎刃而解了。

3.5.1 从某一分机上呼出

如果网关测试正常，你就可以配置从你的 SIP 软电话或 portaudio 呼出了。由于我们是把 FreeSWITCH 当作 PBX 用，所以需要选一个出局字冠。常见的 PBX 一般是内部拨小号，打外部电话就需要加拨 0 或先拨 9 。当然，这是你自己的交换机，你可以用任何你喜欢的数字（甚至是字母）。继续修改拨号计划，创建一个新的 XML 文件—— conf/dialplan/default/call_out.xml，内容如下：

```
<include>
  <extension name="call out">
    <condition field="destination_number" expression="^0(\d+)$">
      <action application="bridge" data="sofia/gateway/gw1/$1"/>
    </condition>
  </extension>
</include>
```

其中，“^0(\d+)$”为正则表达式，“(\d+)”匹配 0 后面的所有数字并存到变量 $1 中。然后通过 bridge 程序通过网关 gw1 打出该号码。当然，建立该 XML 后需要在控制台中执行 reloadxml 使之生效。

⊖ 当添加网关对外注册时，FreeSWITCH 就相当于一个 SIP 客户端，读者可以对比一下，看与配置 X-Lite 有何不同。

3.5.2 呼入电话处理

如果你的 SIP 网关支持 DID[1]，那么你需要知道呼入的 DID 号码。一般来说，呼入的 DID 就是你的 SIP 号码，如果你不知道，也没关系，学习完第 6 章你就能学到如何取得这个值了。创建以下 XML 文件并放到 conf/dialplan/public/my_did.xml 中：

```
<include>
  <extension name="public_did">
    <condition field="destination_number" expression="^(你的 DID)$">
      <action application="transfer" data="1000 XML default"/>
    </condition>
  </extension>
</include>
```

在 FreeSWITCH 中执行 reloadxml 使之生效。上述配置会将来话直接转接到分机 1000 上。在后面的章节中你会学到如何更灵活地处理呼入电话，如转接到语音菜单或语音信箱等。

3.6 小结

本章涵盖了 FreeSWITCH 在 Windows、Linux、MacOSX 三大平台上从安装、配置到调试、使用等相关内容。

如果你能顺利走到这儿，则说明你对 FreeSWITCH 已经爱不释手了。如果你卡在了某处，或某些功能未能实现，这也不是你的错，主要是因为 FreeSWITCH 博大精深，笔者不能在短短的一章内把所有的方面解释清楚。在后面的章节中，你会学到更多的基本概念，从而更加深入地了解 FreeSWITCH 的哲学，也会学到更多的调试技术和技巧，那时解决任何问题都会是小菜一碟了。

最后需要注意的一个问题是，由于版权的限制或某些其他原因，FreeSWITCH 在从源代码编译安装时会从网上自动下载相关的工具、第三方库以及声音文件等（尤其是在 Windows 系统上）。因此，如果编辑安装时没有互联网环境，可能会出现很多奇怪的错误。这时候，笔者建议初学者在有互联网环境的条件下编译，并保证能顺利连接 http://files.freeswitch.org[2]。如果你已经对 FreeSWITCH 比较熟悉了，又必须要在内网环境下编译，可以事先将需要的文件下载好，并放到对应的位置。通过对比有互联的环境下的编译日志能够找到这些位置，关于在不能访问互联网的情况下的编译和安装留给有兴趣的读者自己去练习，在此我们就不多介绍了。

---

[1] Direct Inbound Dial，即对内直接呼入。

[2] files.freeswitch.org 由 CDN 提供，因此在世界各地可能连接到不同的 IP 地址。如果不能正常连接时可以尝试找出你所处区域解析到的 IP 地址，并向官方说明该问题。或者，可以尝试使用其他的 DNS 服务器，如 Google 的 8.8.8.8 及 4.4.4.4 等。

第 4 章
运行 FreeSWITCH

在对 FreeSWITCH 有了一个初步的了解后，各位读者可能已经迫切地想实验它强大的功能了。从这一章起，我们就从最初的运行开始，一步一步进入 FreeSWITCH 的神秘世界。

4.1 命令行参数

一般来说，FreeSWITCH 不需要任何命令行参数就可以启动，但在某些情况下，你需要以一些特殊的参数启动。在此，仅作简单介绍。使用 freeswitch -h 或 freeswitch -help 或 freeswitch --help 会显示以下信息（其中中文部分是作者加的注释）：

```
# freeswitch -help
Usage: freeswitch [OPTIONS]

These are the optional arguments you can pass to freeswitch:
-nf                     -- 不允许 Fork 新进程
-u [user]               -- 启动后以非 root 用户 user 身份运行
-g [group]              -- 启动后以非 root 组 group 身份运行
-help                   -- 显示本帮助信息
-version                -- 显示版本信息
-waste                  -- 允许浪费内存地址空间，FreeSWITCH 仅需 240KB 的栈空间，你可以
                           使用 ulimit -s 240 限制栈空间使用，或使用该选项忽略警告信息
-core                   -- 出错时进行内核转储
-rp                     -- 开启高优先级（实时）设置
-lp                     -- 开启低优先级设置
-np                     -- 普通优先级
-vg                     -- 在 valgrind 下运行，调试内存泄露时使用
-nosql                  -- 不使用 SQL，show channels 类的命令将不能显示结果
-heavy-timer            -- 更精确的时钟。可能会更精确，但对系统要求更高
-nonat                  -- 如果路由器支持 uPnP 或 NAT-PMP，则 FreeSWITCH
                           可以自动解决 NAT 穿越问题。如果路由器不支持，则该选项可以使启动
                           更快
-nocal                  -- 关闭时钟核准。FreeSWITCH 理想的运行环境是 1000Hz 的内核时钟。
                           如果你的内核时钟小于 1000Hz 或在虚拟机上，可以尝试关闭该选项
-nort                   -- 关闭实时时钟
```

```
-stop                          -- 关闭 FreeSWITCH，它会在 run 目录中查找 PID 文件
-nc                            -- 启动到后台模式，没有控制台
-ncwait                        -- 后台模式，等待系统完全初始化完毕之后再退出父进程，隐含“-nc”
                                  选项
-c                             -- 启动到控制台，默认

Options to  control location of files:
-base [confdir]                -- 指定其他的基准目录，在配置文件中使用 $${base}
-conf [confdir]                -- 指定其他的配置文件所在目录，须与 -log、-db 合用
-log [logdir]                  -- 指定其他的日志目录
-run [rundir]                  -- 指定其他存放 PID 文件的运行目录
-db [dbdir]                    -- 指定其他数据库目录
-mod [moddir]                  -- 指定其他模块目录
-htdocs [htdocsdir]            -- 指定其他 HTTP 根目录
-scripts [scriptsdir]          -- 指定其他脚本目录
-temp [directory]              -- 指定其他临时文件目录
-grammar [directory]           -- 指定其他语法目录
-certs [directory]             -- 指定其他 SSL 证书路径
-recordings [directory]-- 指定其他录音目录
-storage [directory]           -- 指定其他存储目录(语音信箱等)
-sounds [directory]            -- 指定其他声音文件目录
```

虽然 FreeSWITCH 参数众多，但我们常用到的也没有几个。笔者经常用到两个参数，首先，就是将 FreeSWITCH 启动到后台的参数：

```
# freeswitch -nc
```

其次，FreeSWITCH 在启动时默认会启用 uPnP（或 NAT-PMP）协议试图查找你的路由器是否支持并在你的路由器上“打洞”，如果你的路由器不支持该协议，这一步可能耗时较长，因而会影响启动速度。所以，如果你只是在内网测试并且像作者一样一天启动很多次的话，建议关掉这个选项。关于 NAT 的内容我们会在后面讲到。

```
# freeswitch -nonat
```

上面两个参数也可以组合来用：

```
# freeswitch -nc -nonat
```

4.2 系统启动脚本

在学习调试阶段，你可以将 FreeSWITCH 启动到前台，而系统真正运行时，你可以使用“-nc”参数启动到后台，然后通过查看 log/freeswitch.log 跟踪系统运行情况[⊖]。

一般情况下，将 FreeSWITCH 启动到前台更容易调试，但如果你又不想在每次关闭 Terminal 时停止 FreeSWITCH，可以考虑借助 screen [⊖]或 tmux 来实现。

在真正的生产系统上，一般需要 FreeSWITCH 能跟系统一起启动。在 UNIX 类系统上，

⊖ 你可以用“tail -f”命令实时跟踪，也可以使用“less +F”，或者在 Windows 平台上找一个叫 wintail 的软件。

⊖ screen 是一个终端模块器，它允许在一个终端连接上打开很多虚拟终端，并且能保持终端会话，使用起来非常方便。详情参见 http://wiki.freeswitch.org/wiki/Freeswitch_In_Screen。

启动脚本一般放在 /etc/init.d/ 下。你可以在系统源代码目录下找到不同系统启动脚本 debian/freeswitch.init 及 build/freeswitch.init.*，大家可以根据自己的需求和实际情况参考使用。在 Windows 上，你也可以将 FreeSWITCH 注册为 Windows 服务（使用“FreeswitchConsole -install”命令）。

4.3 判断 FreeSWITCH 是否运行

某些情况下，我们需要知道 FreeSWITCH 是否已经运行，在 UNIX 类系统上可以用下面的方法判断：

1）看进程是否存在。如果进程已经启动，下列命令能列出所有与 FreeSWITCH 相关的进程：

```
# ps aux | grep freeswitch
```

2）看相关端口是否被占用。假设你使用默认的 5060 端口，则使用下述命令可以列出所有与 5060 端口相关的网络连接：

```
# netstat -an | grep 5060
```

或者，在 Linux 平台上，能直接得到 FreeSWITCH 的进程号（需要 root 权限）：

```
# netstat -anp | grep 5060
```

Windows 平台上也有 netstat 命令，但没有 grep。进程信息可以通过按 Ctrl+Alt+Del 在程序管理器中查看。

4.4 控制台与命令客户端

系统不带参数会启动到控制台，在控制台上可以输入各种命令以控制或查询 FreeSWITCH 的状态。可以尝试输入以下命令（注意，“--”后面是笔者加的注释，它们是相关命令的功能说明，不需要输入）：

```
version                -- 显示当前版本
status                 -- 显示当前状态
sofia status            -- 显示 sofia 状态
help                   -- 显示帮助
```

示例如下：

```
freeswitch> version

FreeSWITCH Version 1.5.2b+git~20130622T184626Z~a38ea77fa3 (git a38ea77 2013-
06-22 18:46:26Z)

注：这里显示系统版本号为 1.5.2b，后面更详细的信息说明 git 版库中对应的版本及提交时间

freeswitch> status
```

```
UP 0 years, 0 days, 12 hours, 23 minutes, 4 seconds, 711 milliseconds, 696 microseconds
FreeSWITCH (Version 1.5.2b git a38ea77 2013-06-22 18:46:26Z) is ready
7 session(s) since startup
0 session(s) - 0 out of max 30 per sec
1000 session(s) max
min idle cpu 0.00/100.00
Current Stack Size/Max 240K/8192K
```

注：该命令显示系统状态。第 1 行说明系统启动了多长时间；第 2 行主要显示了版本号；第 3 行说明系统自启动以来已经处理了 7 个 Session；第 4 行说明当前一个有多少 Session，其中，“0 out of max 30 per sec”说明系统最大支持每秒钟处理 30 个 Session（也就是通常所说的 cps – Call Per Second，这个值可以在配置文件中更改），目前每秒的 Session 数为 0；第 5 行说明系统最大允许多少个并发的 Session（类似 cps，该值也可以修改）；第 6 行显示最小及当前的空闲 CPU；第 7 行显示当前使用的堆栈空间以及系统预留的堆栈地址空间。

为了调试方便，FreeSWITCH 还在 conf/autoload_configs/switch.conf.xml 中定义了一些控制台快捷键。可以通过 F1~F12 这几个按键来使用它们（不过，在某些操作系统上，有些快捷键可能与操作系统相冲突，这时你就只能直接输入这些命令或重新定义它们了），也可以修改配置文件加入比较常用的命令（修改完毕后记着运行 reloadxml 命令使之生效），默认的配置如下：

```
<cli-keybindings>
    <key name="1" value="help"/>
    <key name="2" value="status"/>
    <key name="3" value="show channels"/>
    <key name="4" value="show calls"/>
    <key name="5" value="sofia status"/>
    <key name="6" value="reloadxml"/>
    <key name="7" value="console loglevel 0"/>
    <key name="8" value="console loglevel 7"/>
    <key name="9" value="sofia status profile internal"/>
    <key name="10" value="sofia profile internal siptrace on"/>
    <key name="11" value="sofia profile internal siptrace off"/>
    <key name="12" value="version"/>
</cli-keybindings>
```

FreeSWITCH 是一个典型的 Client/Server 结构，不管 FreeSWITCH 运行在前台还是后台，你都可以使用客户端软件 fs_cli 连接 FreeSWITCH。

fs_cli 是一个类似 Telnet 的客户端（也类似于 Asterisk 中的“asterisk -r”命令），它使用 FreeSWITCH 的 ESL（Event Socket Library）协议与 FreeSWITCH 通信。当然，使用该协议需要加载模块 mod_event_socket，该模块是默认加载的。

正常情况下，直接输入 bin/fs_cli 即可连接到 FreeSWITCH 上，并出现系统提示符。如果出现类似：

```
[ERROR] libs/esl/fs_cli.c:652 main() Error Connecting [Socket Connection Error]
```

这样的错误，说明 FreeSWITCH 没有启动或 mod_event_socket 没有正确加载，请检查 TCP 的 8021 端口是否处于监听状态或被其他进程占用。

fs_cli 也支持很多命令行参数，值得一提的是 -x 参数，它允许执行一条命令后退出，这在编写脚本程序时非常有用（如果它能支持管道会更有用，但是它不支持）：

```
$ bin/fs_cli -x "version"                                    # 显示版本号
$ bin/fs_cli -x "status"                                     # 显示状态
$ bin/fs_cli -x "originate user/1000 &bridge(user/1001)"     # 回拨
```

下面让我们看一个使用示例：

```
$ fs_cli -x "version"
FreeSWITCH Version 1.5.2b+git~20130622T184626Z~a38ea77fa3 (git a38ea77 2013-06-22 18:46:26Z)
```

其他的参数都可以通过配置文件来实现，在这里就不多说了。可以参见 http://wiki.freeswitch.org/wiki/Fs_cli。

使用 fs_cli，不仅可以连接到本机的 FreeSWITCH，也可以连接到其他机器的 FreeSWITCH 上（或本机另外的 FreeSWITCH 实例上），通过在用户主目录下编辑配置文件 .fs_cli_conf（注意前面的点“.”），可以定义要连接的多个机器：

```
[server1]
host     => 192.168.1.10
port     => 8021
password => secret_password
debug    => 7

[server2]
host     => 192.168.1.11
port     => 8021
password => someother_password
debug    => 0
```

注意，如果要连接到其他机器，要确保目标机器的 FreeSWITCH 的 Event Socket 是监听在真实网卡的 IP 地址上，而不是 127.0.0.1，这可以通过将 conf/autoload_configs/event_socekt.conf.xml 中的 IP 地址改成为服务器 IP 或“0.0.0.0”实现，当然，这可能带来潜在的安全性问题。如果你的服务器运行在公网上，则需要考虑你是否确实需要这样做，或者至少考虑设置一下 ACL 或防火墙规则只允许特定的 IP 地址访问。当然，记得改完后在控制台上要执行“reload mod_event_socket”。另外，在 UNIX 中，以点开头的文件是隐藏文件，普通的“ls”命令是不能列出它的，可以使用“ls -a”列出这些文件。

一旦配置好，就可以这样使用它：

```
$ fs_cli server1
$ fs_cli server2
```

在 fs_cli 中，有几个特殊的命令，它们是以“/”开头的，这些命令并不直接发送到 FreeSWITCH，而是先由 fs_cli 处理。/quit、/bye、/exit、Ctrl + D 都可以退出 fs_cli；/help 是帮助。

其他一些以“/”开头的指令与 Event Socket 中相关的命令相同，如：

```
/event       -- 开启事件接收
/noevents    -- 关闭事件接收
/nixevent    -- 除了特定一种外，开启所有事件
/log         -- 设置 log 级别，如 /log info 或 /log debug 等
```

```
/nolog        -- 关闭 log
/filter       -- 过滤事件
```

除此之外，其他命令都与直接在 FreeSWITCH 控制台上执行是一样的。在 fs_cli 中也支持快捷键，最常用的快捷键是 F6（reloadxml）、F7（关闭 log 输出）、F8（开启 debug 级别的 log 输出）。

在 UNIX 类平台上，FreeSWITCH 控制台和 fs_cli 两者都通过 libeditline 支持命令行编辑功能。可以通过上、下箭头查看历史命令。

4.5 呼叫

呼叫是 FreeSWITCH 最基本的功能，下面我们一起来看一下 FreeSWITCH 中与呼叫相关的命令和概念。

4.5.1 发起呼叫

可以在 FreeSWITCH 中使用 originate 命令发起一次呼叫。下面我们看个例子。假设用户 1000 已经注册，那么可以运行如下命令：

```
freeswitch> originate user/1000 &echo
```

上述命令在呼叫 1000 这个用户后，便执行 echo 这个程序。echo 是一个回音程序，即它会把任何它“听到”的声音（或视频）再返回（说 / 播）给对方。因此，如果这时候用户 1000 接了电话，无论说什么都会听到自己的声音。

4.5.2 呼叫字符串

在 4.5.1 节的例子中，“user/1000”称为呼叫字符串（Dial String，有时也叫 Call URL）。“user”是一种特殊的呼叫字符串，在后面我们还会看到其他的呼叫字符串。

下面让我们来看一个例子。假设 FreeSWITCH UA 的地址为 192.168.4.4:5050，alice UA 的地址为 192.168.4.4:5090，bob UA 的地址为 192.168.4.4:26000。若 alice 已向 FreeSWITCH 注册，在 FreeSWITCH 中就可以看到 alice 的注册信息：

```
freeswitch> sofia status profile internal reg

Registrations:
=================================================================================
Call-ID:         ZTRkYjdjYzY0OWFhNDRhOGFkNDUxMTdhMWJhNjRmNmE.
User:            alice@192.168.4.4
Contact:          "Alice" <sip:alice@192.168.4.4:5090;rinstance=a86a656037ccfaba
;transport=UDP>
Agent:           Zoiper rev.5415
Status:          Registered(UDP)(unknown) EXP(2010-05-02 18:10:53)
Host:            du-sevens-mac-pro.local
IP:              192.168.4.4
```

```
Port:              5090
Auth-User:         alice
Auth-Realm:        192.168.4.4
MWI-Account:       alice@192.168.4.4

=====================================================================
```

FreeSWITCH 根据 Contact 字段知道 alice 的 SIP 地址：sip:alice@192.168.4.4:5090。当使用 originate 命令呼叫 user/alice 这个呼叫字符串时，FreeSWITCH 便会在用户目录中查找 alice 这个用户，找到她的 dial-string 参数[⊖]，dial-string 参数通常包含 alice 实际 Contact 地址的查找方法，FreeSWITCH 进而找到 alice 的 Contact 地址 sip:alice@192.168.4.4:5090 并向其发送 INVITE 请求。

4.6 API 与 App

在 4.5.2 节的例子中，originate 是 FreeSWITCH 内部的一个命令（Command），它用于控制 FreeSWITCH 发起一个呼叫。FreeSWITCH 的命令不仅可以在控制台上使用，也可以在各种嵌入式脚本、Event Socket（fs_cli 就是使用了 ESL 库）或 HTTP RPC 上使用，所有命令都遵循一个抽象的接口，因而这些命令又称 API Commands[⊜]。

echo 则是一个常用的应用程序（Application，App），它的作用是控制一个 Channel 的一端。我们知道，一个 Channel 有两端，在上面的例子中，alice 是一端，另一端就是 echo。电话接通后相当于 alice 在跟 echo 通话。后面我们会讲到，它们实际上组成了 FreeSWITCH 的一条腿（leg），这种通话称为“单腿通话（one-legged connection）”。

另一个常用的 App 是 park，其使用格式如下：

```
freeswitch> originate user/alice &park
```

当我们初始化了一个呼叫时，在 alice 接电话后对端必须有一个人在跟其讲话（否则，如果一个 Channel 只有一端，那是不可思议的事情），而如果这时 FreeSWITCH 找不到一个合适的人跟 alice 通话，那么它可以将该电话“挂起”，park 便是实现这个功能，它相当于一个 Channel 特殊的一端[⊜]。

park 的用户体验并不好，因为 alice 不知道要等多长时间才有人接电话，由于它听不到任何声音，这时它就会奇怪电话到底有没有接通。相对而言，另一个程序 hold 则比较友好，它能在等待的同时播放保持音乐（Music on Hold，MOH）。hold 的使用格式如下：

⊖ 该参数可以在默认配置文件中的 conf/directory/default.xml 中找到，当然它比较复杂，我们将在后面的章节中再深入讨论。

⊜ 当 API 与 App 相对时，我们一般叫 API，这个名字来源于它的具体实现，在 FreeSWITCH 内部的实现中，所有的 API 都来源于一个叫 switch_api_interface_t 的结构。在其他情况下，我们不是刻意比较它与 App 的区别时，更倾向于称之为“命令”，这样描述起来更自然一些。

⊜ 笔者常举的一个例子是：如果把一个 Channel 看作是一根绳子，那么该例中是 alice 相当于拉着绳子的一头，而对端则拴（park）在一枚钉子上。

```
freeswitch>originate user/alice &hold
```

当然，你也可以直接播放一个特定的声音文件，如下：

```
freeswitch> originate user/alice &playback(/root/welcome.wav)
```

或者直接录音，如下：

```
freeswitch> originate user/alice &record(/tmp/voice_of_alice.wav)
```

以上的例子实际上都只是建立一个 Channel，相当于 FreeSWITCH 作为一个 UA 跟 alice 通话。它是个一条腿（只有 a-leg）的通话。在大多数情况下，FreeSWITCH 都是作为一个 B2BUA 来桥接两个 UA 进行通话的。在 alice 接听电话以后，bridge 程序可以再启动一个 UA 呼叫 bob，如下：

```
freeswitch> originate user/alice &bridge(user/bob)
```

至此，alice 和 bob 终于可以通话了。我们也可以用另一种方式建立他们之间的通话，具体步骤如下：

```
originate user/alice &park
originate user/bob &park
show channels
uuid_bridge <alice_uuid> <bob_uuid>
```

在这里，我们分别呼叫 alice 和 bob，并把他们暂时 park 到一个地方。通过命令 show channels 我们可以知道每个 Channel 的 UUID，然后使用 uuid_bridge 命令将两个 Channel 桥接起来，而上一种方式实际上是先桥接，再呼叫 bob。

上面我们一共学习了两条命令（API）——originate 和 uuid_bridge，以及几个程序（App）——echo、park、bridge 等。细心的读者可能会发现，uuid_bridge API 和 bridge App 有些类似。我们知道它们一个是先呼叫后桥接，另一个是先桥接后呼叫，那么它们到底有什么本质的区别呢？

简单来说，一个 App 是一个程序（Application），它作为一个 Channel 一端与另一端的 UA 进行通信，相当于它工作在 Channel 内部；而一个 API 则是独立于一个 Channel 之外的，它只能通过找到 Channel 的 UUID 来控制一个 Channel（如果需要的话），相当于一个第三者。这就是 API 与 App 最本质的区别。⊖

通常我们在控制台上输入的命令都是 API；而在 dialplan 中执行的程序都是 App(dialplan 中也能执行一些特殊的 API)。大部分公用的 API 都是在 mod_commands 模块中加载的；而 App 则在 mod_dptools 中，因而 App 又称为拨号计划工具（Dialplan Tools）。某些模块（如 mod_sofia）有自己特有的 API 和 App。

某些 App 有与其对应的 API，如上述的 bridge 和 uuid_bridge，还有 transfer 和 uuid_transfer、playback 和 uuid_playback 等。uuid 一族的 API 都是在一个 Channel 之外对 Channel 进行控制的，它们应用于不能参与到通话中却又想对正在通话的 Channel 做点什

⊖ 当然，除 originate 以及 uuid 一族的 API(以 uuid_ 开头的 API) 外，其他大部分 API(如 version,、status 等) 一般是没有相对应的 App 的，因而这里就没有讨论的必要了。

么的场景中。例如 alice 和 bob 正在畅聊，有个“坏蛋”使用 uuid_kill 将电话切断，或使用 uuid_broadcast 对他们广播恶作剧音频，或者使用 uuid_record 对他们谈话的内容录音等。

关于 API 和 App 更详细的内容，可以参见：

- mod_dptools: http://wiki.freeswitch.org/wiki/Mod_dptools。
- mod_commands: http://wiki.freeswitch.org/wiki/Mod_commands。

4.7 API 命令帮助

在本章的最后，我们来学习一下如何使用 FreeSWITCH 的 API 命令帮助。使用 help 可以列出所有命令的帮助信息，具体示例如下：

```
freeswitch> help

Valid Commands:

...,,Shutdown,mod_commands
acl,<ip><list_name>,Compare an ip to an acl list,mod_commands
alias,[add|stickyadd] <alias><command> | del [<alias>|*],Alias,mod_commands
banner,,Return the system banner,mod_commands
bg_system,<command>,Execute a system command in the background,mod_commands
bgapi,<command>[ <arg>],Execute an api command in a thread,mod_commands
break,<uuid> [all],uuid_break,mod_commands
chat,<proto>|<from>|<to>|<message>|[<content-type>],chat,mod_dptools
cluechoo,syntax,Cluechoo API,mod_cluechoo
complete,add <word>|del [<word>|*],Complete,mod_commands
cond,<expr> ? <true val> : <false val>,Evaluate a conditional,mod_commands
conference ...
```

以 bgapi 为例，它的作用是把命令放到后台执行（命令执行都是阻塞的，有些命令需要执行相对较长的时间才能返回，因而有时我们会把某些命令放到后台执行）。其后面的 <command> 表示是一个必选参数，它的值可以是除 bgapi 以外的任何其他命令；[<arg>] 放到方括号中，表示是一个可选参数，该参数是实际的 <command> 的参数，由于有些命令是没有参数的，因而它是可选的。

某些命令也有自己更详细的帮助信息，如 sofia：

```
freeswitch> sofia help

USAGE:
--------------------------------------------------------------------------------
sofia help
sofia profile <profile_name> [[start|stop|restart|rescan]
    [reloadxml]|flush_inbound_reg [<call_id>] [reboot]|[register|unregister]
....
```

其中，用尖括号（<>）括起来的表示是要输入的参数；而用方括号（[ ]）括起来的则表示为可选项，该参数可以有也可以没有；用竖线（|）分开的参数列表表示“或”的关系，即

只能选其一。

FreeSWITCH 的命令参数没有统一的解析函数，都是由命令本身的函数负责解析的，因而不是很规范，不同的命令可能有不同的风格，所以使用时，除使用帮助信息外，最好还是查阅一下 Wiki 上的帮助。

4.8 小结

本章介绍了如何启动与控制 FreeSWITCH，并提到了几个常用的命令。另外，本章还着重讲述了 App 与 API 的区别、呼叫字符串的概念等，搞清楚这些概念对后面的学习是很有帮助的。

我们无法在一本书中讲解所有的命令，因此，我们也在本章中介绍了如何实时获取在线帮助，了解了获取帮助的方法和相关规律，就可以随时迅速学习和掌握新的命令和知识了。

第 5 章

FreeSWITCH 架构

通过对上一章的学习，我们了解了 FreeSWITCH 是如何启动和运行的。在本章，我们一起来看一看 FreeSWITCH 的内部架构。

5.1 总体架构

总的来说，FreeSWITCH 由一个稳定的核心（Core）及一些外围模块组成。这些外围的模块根据其功能和用途的不同又分为 Endpoint、Codec、Dialplan、Application 等不同的类别，如图 5-1 所示。其中，数据库的代码是在核心中实现的，图 5-1 中之所以将核心单独列出来只是因为从逻辑上看它是独立的⊖。

FreeSWITCH 内部使用线程模型来处理并发请求，每个连接都在单独的线程中进行处理，不同的线程间通过 Mutex 互斥访问共享资源，并通过消息和异步事件等方式进行通信。这种架构能处理很高的并发，并且在多核环境中运算能均匀地分布到多颗 CPU 或单 CPU 的多个核心上。FreeSWITCH 的核心非常短小精悍，这也是其保持稳定的关键。绝大部分应用层的功能都在外围的模块中实现。外围模块是可以动态加载（以及卸载）的，在实际应用中可以只加载用到的模块。外围模块通过核心提供的 Public API 与核心进行通信，而核心则通过回调（或称钩子）机制执行外围模块中的代码。

5.1.1 核心

FreeSWITCH 的核心是 Core，它包含了关键的数据结构和复杂的代码、状态机、数据库等，这些代码只出现在核心中，并保持了最大限度的抽象和重用。外围模块只能通过核心代码提供的公共应用程序接口（Public API）调用核心的功能，因而核心运行在一个受保护的环境中，核心代码都是经过了精心编写和严格测试的，最大限度地保证了系统整体的稳定。

⊖ 除此之外，核心也实现了一些逻辑上应该在外部模块中实现的接口，如编解码接口中的 PCMU 和 PCMA。

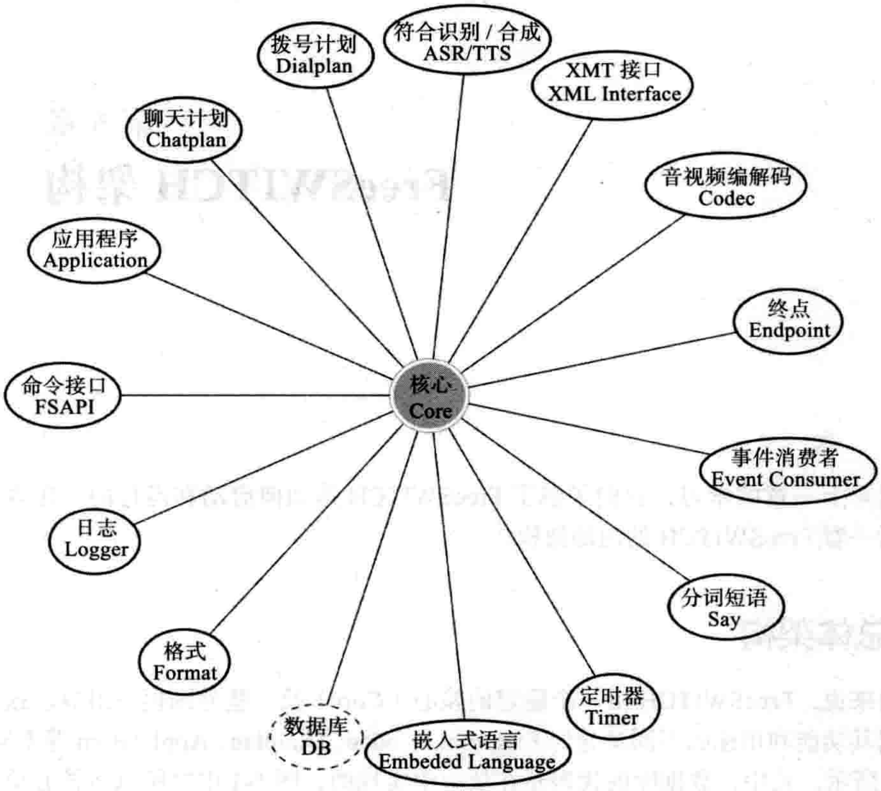


图 5-1 FreeSWITCH 架构示意图

核心代码保持了最高程度的抽象，因而它可以调用不同功能、不同协议的模块。同时，实现良好的 API 也使得编写不同的外围模块非常容易。

1. 数据库（DB）

FreeSWITCH 的核心除了使用内部的队列、哈希表存储数据外，也使用外部的关系型数据库存储数据。使用外部数据库的好处是查询数据不用锁定内存数据结构，这不仅能提高性能，还能降低死锁的风险，保证了系统稳定。命令 show calls、show channels 等都可直接从数据库中读取内容并显示。

FreeSWITCH 支持多种流行的关系型数据库。为了尽量减少对其他系统的依赖，FreeSWITCH 默认使用的数据库类型是 SQLite。SQLite 是一种嵌入式数据库，FreeSWITCH 可以直接调用它提供的库函数来访问数据。由于 SQLite 会进行读锁定，因此在使用 SQLite 时不建议通过外部应用直接读取核心数据库[⊖]。

⊖ 如果使用了 ODBC 连接其他数据库如 PostgreSQL 及 MySQL 时，则又建议直接读取数据库的内容获取信息。

除 SQLite 外，系统也支持通过使用 ODBC ⊖方式连接其他数据库，如 PostgreSQL、MySQL 等（有的人也成功地使用 ODBC 方式连接到 Oracle 及 MS SQL Server 等，但可能需要改动一些代码）。由于 PostgreSQL、MySQL 等都是 Client/Server 的结构，允许多个客户端连接查询，因此外部程序可以直接查询数据库，获取当前分机的注册状态及通话状态等。

另外，FreeSWITCH 内部也提供了一个对 PostgreSQL 数据库的原生支持。它可以通过 PostgreSQL 提供的客户端库直接访问 PostgreSQL 服务，减少了中间的 ODBC 环节。

FreeSWITCH 使用一个核心的数据库（默认的存放位置是 /usr/local/freeswitch/db/core.db）来记录系统的接口（interfaces）、任务（tasks）以及当前的通道（channels）、通话（calls）等实时数据。某些模块，如 mod_sofia、mod_fifo 等，都有自己的数据库（表），一般来说，这些模块也提供相关的 API 用于从这些表里查询数据。必要时，用户也可以直接查询这些数据库（表）来获取数据（如 mod_sofia 中的分机注册数据等）。

系统对数据库操作做了优化，在高并发状态下，核心会尽量将几百条 SQL 语句一起执行，这可以大大提高系统性能。

2. 公共应用程序接口（Public API）

FreeSWITCH 在核心层实现了一些 Public API。这些 Public API 可以被外围的模块调用。例如，当 FreeSWITCH 外围的 Endpoint 模块收到一个呼入请求时，该模块就可以调用核心的 switch_core_session_request 函数为该呼叫生成一个新的 Session，此后，该呼叫的生命周期就由该 Session 管理。如果呼叫挂断，就调用 switch_core_session_destroy 函数将该 Session 释放。

一般来说，与 Session 相关的函数都是与呼叫相关的。进一步细分，我们可以认为它主要与信令层相关，因为一个呼叫的生命周期是由信令来控制的。为了便于理解，接下来我们再举一个媒体层的例子。我们在 1.5 节中讲到，在 SIP 通话中，实际的语音、视频数据是使用 RTP 协议传输的（在这里为了简单起见我们只讨论语音）。为了在通话中使通话双方能彼此听到对方的声音，需要在已经建立的 Session 的基础上再增加一个 RTP 的 Session，通过它进行 RTP 媒体流的收发等。使用 switch_rtp_new 函数可以建立一个新的 RTP Session，在挂断电话前，可以使用 switch_rtp_destroy 来释放它。

核心的代码是任何模块都可以调用的，所以不同的 Endpoint 模块都可以调用核心中与 Session 相关的函数，以创建或释放一个 Session，以及创建或释放相关的媒体流。当然，除了 Session、RTP 等与呼叫相关的函数外，核心中还有一些通用的工具函数，如生成 JSON 格式的函数、通用字符串处理函数、与通用数据数据结构相关的函数等。这些工具函数提供统一的接口，这方便开发外围模块，也有助于统一代码的风格，利于修改和维护。关于这些函数我们将在第 20 章讲解 FreeSWITCH 代码导读时再详细分析，在此就不多费笔墨了。

⊖ 如果在 UNIX 类系统上使用 ODBC，则需要安装 UnixODBC，并进行正确的配置（我们将在本书第二部分详细讲解 ODBC 的配置和使用）。如果要编译安装，还需要开发包 unixodbc-devel（CentOS）或 unixodbc-dev（Debian/Ubuntu）。

3. 接口（Interface）

FreeSWITCH 在核心中除实现了大量的 Public API 供外围模块调用外，还提供了很多抽象的接口，如图 5-1 中所示的 Endpoint、Codec 等。这些接口对同类型的逻辑或功能实体进行了抽象，但没有具体实现。具体的实现一般由外围的模块负责，核心层通过回调（钩子）方式调用具体的实现代码或函数[⊖]。

FreeSWITCH 核心层定义了以下接口（来自 switch_types.h，注释为作者所加）：

```
typedef enum {
    SWITCH_ENDPOINT_INTERFACE,          // 终点
    SWITCH_TIMER_INTERFACE,             // 定时器
    SWITCH_DIALPLAN_INTERFACE,          // 拨号计划
    SWITCH_CODEC_INTERFACE,             // 编解码
    SWITCH_APPLICATION_INTERFACE,       // 应用程序
    SWITCH_API_INTERFACE,               // 命令
    SWITCH_FILE_INTERFACE,              // 文件
    SWITCH_SPEECH_INTERFACE,            // 语音合成
    SWITCH_DIRECTORY_INTERFACE,         // 用户目录
    SWITCH_CHAT_INTERFACE,              // 聊天计划
    SWITCH_SAY_INTERFACE,               // 分词短语
    SWITCH_ASR_INTERFACE,               // 语音识别
    SWITCH_MANAGEMENT_INTERFACE,        // 网管接口
    SWITCH_LIMIT_INTERFACE,             // 资源限制接口
    SWITCH_CHAT_APPLICATION_INTERFACE // 聊天应用程序接口
} switch_module_interface_name_t;
```

具体的，外围模块可以选择实现其中一个或多个接口，并向核心层“注册”这些接口。核心层在需要这些接口时，会回调在这些接口中约定的回调函数。举例来说，在 SIP 通话中，如果核心需要取音频数据，它会调用 mod_sofia 中的相关回调函数，最后 mod_sofia 将调用核心中与 RTP 相关的函数，以便从 RTP 媒体流中取得数据；而在 mod_portaudio 中，音频数据是从本地声卡中获得的。也就是说，在上层应用中，我们可能只需读取一段音频数据，而不必关心该音频是怎么产生的或从哪里来的。在此，上层应用可以调用核心中的 switch_core_session_read_frame 函数以便从相关的 Session 中取得一“帧”音频数据，但该数据具体是从 RTP 流中还是本地声卡中取得，是由具体的 Endpoint 决定的，上层应用不知道也不关心它到底是怎么来的。

通过这种方式，在上层屏蔽了下层实现方式的细节，保证了上层实现接口的简单和稳定。同时，很容易通过这种机制扩展其他类型的模块，进而支持越来越多的协议、功能和特性。

4. 事件（Event）

除了使用 Public API 及接口回调方式执行内部逻辑和通信外，FreeSWITCH 在内部也使用消息和事件机制进行进程间和模块间通信。消息机制完全是内部的，在此我们就不多讲了。事件机制既可在内部使用，也可在外部使用。当 FreeSWITCH 内部状态发生变化或收到一些新的消息时，会产生（Fire）一些事件。事件机制是一种“生产者 – 消费者”模

⊖ 可以想象为面向对象的语言中的基类或纯虚函数，我们将在本书的第三部分深入剖析。

型。事件的产生和处理（生产和消费）是异步的，是一种松耦合的关系。这些事件可以在 FreeSWITCH 内部通过绑定（Bind）一定的回调函数进行捕获，即 FreeSWITCH 的核心事件系统会依次回调这些回调函数，完成相应的功能。另外，在嵌入式脚本（如 Lua）中也可以订阅相关的事件并进行处理。

在 FreeSWITCH 外部，也可以通过 Event Socket 等接口订阅相关的事件，通过这种方式可了解 FreeSWITCH 内部发生了什么，如当前呼叫的状态等。fs_cli 就是一个典型的外部程序，它通过 Event Socket 与 FreeSWITCH 通信，可以对 FreeSWITCH 进行控制和管理，也可以订阅相关的事件对 FreeSWITCH 的运行情况进行监控。订阅事件最简单的方法是：

```
fs_cli> /event plain ALL
```

在 fs_cli 中执行上述命令可以订阅所有的事件。FreeSWITCH 的事件主要有两大类：一类是主事件，可根据事件的名字（Event-Name）来区分，如 CHANNEL_ANSWER（应答）、CHANNEL_HANGUP（挂机）等；另一类是自定义事件，它们的 Event-Name 永远是 CUSTOM，不同的事件类型可根据子类型（Event-Subclass）来区分，如 sofia::register（SIP 注册）、sofia::unregister（SIP 注销）等。

在 fs_cli 中可以单独订阅某类事件，如：

```
fs_cli> /event plain CHANNEL_ANSWER
fs_cli> /event plain CUSTOM sofia::register
```

在此，读者对事件有一个简单的了解就可以了，我们将在后面的章节中更详细地讨论事件机制、函数及相关用法。

5.1.2 接口实现

从上一节中我们了解到，接口都是抽象的，其中只有少量在核心中有具体实现（如 PCM 编解码的实现），大部分最终由外部的模块来实现。与接口的抽象概念相比，我们更关注具体实现。因此，在本节中我们把接口和模块结合在一起来讲。由于接口和模块的分类及源代码结构并不总是一一对应的，因此我们在这里不关注这些概念的不同点，而专注于它们的共同点。下面我们简单介绍一下这些不接口以及相关的模块：

- **终点（Endpoint）**。Endpoint 是终结 FreeSWITCH 的地方，也就是说再往外走就超出 FreeSWITCH 的控制了。它主要包含了不同呼叫控制协议的接口，如 SIP、TDM 硬件、H323 以及 Google Talk 等。这使得 FreeSWITCH 可以与众多不同的电话系统进行通信。例如，可以使用 mod_skypopen 与 Skype 网络进行通信。另外，我们在 3.4 节也讲过，它还可以通过 mod_portaudio 驱动本地声卡，用作一个软电话。
- **拨号计划（Dialplan）**。Dialplan 主要提供查找电话路由功能。系统默认的 Dialplan 由 mod_dialplan_xml 提供，它是以 XML 描述的。除此之外，FreeSWITCH 也支持 Asterisk 风格的配置文件以及 ENUM 查询等。
- **聊天计划（Chatplan）**。类似于 Dialplan，不同的是 Chatplan 主要对文本消息进行路由，如 SIP SIMPLE、Skype Message、XMPP Message 等。它是在 mod_sms 中实现的。

- **应用程序（Application，APP）**。FreeSWITCH 提供了许多 App 使复杂的任务变得异常简单，如 mod_voicemail 模块可以很简单地实现语音留言；而 mod_conference 模块则可以实现高质量的多方会议。我们在本书中多次提到 FreeSWITCH 本身是一个 B2BUA，而实际上所有与 FreeSWITCH 的通话（或通过 FreeSWITCH 的通话）都是在与一个或多个 App 在交互。mod_dptools 模块提供了系统中大部分的 App。
- **命令接口（FSAPI）**[⊖]。FSAPI 简称 API，它是一种对外的命令接口，它的原理非常简单——输入一个简单的字符串（以空格分隔），该字符串由模块的内部函数处理，然后得到一个输出。输出可以是一个简单的字符串、一大串文本、XML 或 JSON 文本。通过使用 FSAPI，一个模块可以通过命令字符串的方式调用另一个模块提供的功能，而不用连接其他模块的函数（或代码，实际上 FreeSWITCH 不鼓励也不支持模块间互相连接）。它实际上是一种高度抽象的输入 / 输出机制，不仅在模块内部，在 FreeSWITCH 外部也可能通过 Event Socket（第 3 章所说的 fs_cli 就是用这种方式）或 XML-RPC 等进行调用。系统中大部分的 API 都是由 mod_commands 模块提供的，有的模块实现了一些与本模块相关的 API。
- **XML 接口（XML Interface）**。XML 接口支持多种获取 XML 的方式，它可以从本地的配置文件或数据库中读取，甚至可以从一个能动态返回 XML 的远程 HTTP 服务器中读取。对 XML 的解析和访问是在核心中实现的，但对 XML 的应用和扩展都是在外部模块中完成的，如 mod_xml_rpc、mod_xml_curl、mod_xml_cdr 等。
- **编解码器（Codec）**。FreeSWITCH 支持最广泛的 Codec，除了大多数 VoIP 系统支持的 G711、G722、G729、GSM 外，它还支持 iLBC、BV16/32、SILK、iSAC、CELT、OPUS 等。它可以同时桥接不同采样频率的电话以及电话会议等。G711（即 PCM）编解码是在核心中实现的，其他的编解码一般都由独立的模块实现，如 mod_ilbc 实现了 iLBC 编解码。
- **语音识别及语音合成（ASR/TTS）**。支持语音自动识别（ASR）及文本 / 语音转换（TTS）。可以通过本地模块实现（如 mod_flite），也可以通过 MRCP 协议与其他语音产品对接（如 mod_unimrcp）实现。
- **格式、文件接口（Format，File Interface）**。支持不同格式的声音文件回放、录音，如 WAV、MP3 等。mod_sndfile 模块通过 libsndfile 库提供了对大部分音频文件格式的支持。MP3 格式是在 mod_shout 中实现的。
- **日志（Logger）**。日志可以写到控制台、日志文件、系统日志（syslog）以及远程的日志服务器。实现日志功能的模块有 mod_console、mod_logfile、mod_syslog 等。
- **定时器（Timer）**。实时的话音通话需要非常准确的定时器。在 FreeSWITCH 中，可

⊖ 即 FreeSWITCH API，它在 FreeSWITCH 内部是使用 switch_api_interface_t 类型实现的，即它实现的是 FreeSWITCH 的命令行接口。在后面要讲到的嵌入式语言及 ESL 中也使用 API（如 mod_lua 中的 freeswitch.API 及 ESL 中的 api 命令）。注意，不要将这里的 API 与常见的应用程序接口（Application Programming Interface）混淆。参见 4.6 节。

以使用软时钟（soft timer）或内核提供的时钟来定时（如 Linux 中的 timerfd 或 posix timer）。值得一提的是，FreeSWITCH 最理想的工作时钟频率是 1000Hz，而某些 Linux 发行版或虚拟机的内核的默认时钟可能是 100Hz 或 250Hz，在这种情况下，可以使用内核提供的时钟接口，或者可以重新编译内核调整时钟频率。

- **嵌入式语言（Embeded Language）**。通过 swig 可支持多种嵌入式语言进而控制呼叫流程，如 Lua、Javascript、Perl 等。
- **事件套接字（Event Socket）**。通过 Event Socket 可以使用任何其他语言（只要支持 Socket），通过 TCP Socket 可控制呼叫流程、扩展 FreeSWITCH 的功能。

5.2 目录结构

在 UNIX 类系统上，FreeSWITCH 默认的安装位置是 /usr/local/freeswitch，在 Windows 上可能是 C:\Programming Files\FreeSWITCH。在这两种系统上，FreeSWITCH 的目录结构大致相同，如表 5-1 所示。

表 5-1 FreesSWITCH 目录结构

目　录	说　明
bin	可执行程序
db	系统数据库（sqlite），将呼叫信息存放到数据库中，这样在查询时就无须对核心数据结构加锁了
htdocs	HTTP Server 根目录
lib	库文件
mod	可加载模块
run	运行目录，存放 FreeSWITCH 运行时的 PID
sounds	声音文件，使用 playback() 时默认的寻找路径
grammar	语法，用于 ASR
include	头文件
log	日志，CDR 等
recordings	录音，使用 record() 时默认的存放路径
scripts	嵌入式语言写的脚本，如使用 lua( )、luarun( )、jsrun 等默认寻找的路径
storage	语音留言（Voicemail）的录音
conf	配置文件，详见 5.3 节

一般来说，大部分的目录结构都是扁平的，没有什么子目录。比较例外的有以下几个：

- sounds：sounds 目录中存放了各种声音文件。FreeSWITCH 支持多种频率的通话，因而也支持多种频率的声音文件。如下面的列表所示。

```
├── en
│   └── us
│       └── callie
│           ├── ascii
```

```
|        |   └── 8000
|        ├── base256
|        |   └── 8000
|        ├── conference
|        |   └── 8000
|        ├── currency
|        |   └── 8000
|        ├── digits
|        |   └── 8000
|        ├── directory
|        |   └── 8000
|        ├── ivr
|        |   └── 8000
|        ├── misc
|        |   └── 8000
|        ├── phonetic-ascii
|        |   └── 8000
|        ├── time
|        |   └── 8000
|        ├── voicemail
|        |   └── 8000
|        └── zrtp
|            └── 8000
├── music
|   └── 8000
└── zh
    └── cn
        └── link
            ├── ascii
            |   └── 8000
            ├── conference
            |   └── 8000
```

注意

中文目录与英文目录结构大致相同，以下略。

声音文件存放在多级目录中。其中，en 表示英语、us 表示美国英语、callie 表示嗓音（即人的声音，不同的人，音调和音色不同），下面的 ascii、conference 表示声音文件的类别。在这里我们只有 8000Hz 的语音，因此里面只有 8000 这个子目录，具体的声音文件在 8000 目录下面。

music 目录中存放的是 MOH，即保持音乐。

- storage：storage 目录中存放从其他 HTTP 服务器下载过来的语音文件缓存（http_file_cache）以及 voicemail 录音留言等。其中 voicemail 的声音文件也是以 profile（如下面的 default）和 domain（如下面的 192.168.1.123）分级存放的，具体如下：

```
├── http_file_cache
└── voicemail
    └── default
        ├── 192.168.1.123
        |   ├── 1001
        |   └── 1007
        |
```

```
├── 192.168.1.124
│   ├── 1000
│   ├── 1001
│   ├── 1003
│   └── 1004
└── 192.168.127.103
    └── 1001
```

❑ conf: conf 目录中存放的是 FreeSWITCH 的配置文件，我们将在下一节单独说明。

5.3 配置文件

配置文件由许多 XML 文件组成。在系统装载时，XML 解析器会将所有 XML 文件组织在一起，并读入内存，组成一个大的 XML 文档（Document），称为 XML 注册表。XML 文档本身非常适合描述复杂的数据结构，在 FreeSWITCH 中可以非常灵活地使用这些数据。这种设计的好处是可以帮助实现非常高的可扩展性。并且，XML 也是一个国际通用的标准，其他外部应用程序也可以很容易地生成 XML，从而很容易地与 FreeSWITCH 集成。另外，系统还允许在某些 XML 节点上安装回调函数，当这些节点的数据变化时（如在 reloadxml 时），系统便自动调用这些回调函数，以更新系统的内部数据或状态等。

使用 XML 唯一的不足就是手工编辑这些 XML 比较困难，但正如其作者所言[⊖]，他绝对不是 XML 的铁杆粉丝，他认为人们在理解 XML 的过程中迷失了方向，相反，把一个复杂的对象序列化成文本形式以及用一个易于解析又易于生成的标记语言来描述这正是 XML 的长处。使用 XML 的最高境界是让所有人意识不到它的存在。虽然现在还没有谁能做到这一点，但随着 XML 的发展以及更多的 GUI 界面和配置程序出现，终有一天会发展到只有高级用户才会去看这些 XML，那时其他用户就意识不到 XML 的存在了。

在继续讨论配置文件之前，我们先来看一下这些配置文件的目录结构。在本书第 3 章我们已经提到过，配置文件的目录结构如下（其中，带扩展名的是文件，其他的一般都是目录）：

```
├── freeswitch.xml
├── mime.types
├── notify-voicemail.tpl
├── tetris.ttml
├── vars.xml
├── voicemail.tpl
├── web-vm.tpl
├── autoload_configs
├── chatplan
├── dialplan
```

⊖ Trust me. I am far from XML's biggest fan, but I do feel it's actual uses are lost in a sea of contrived forced solutions that give it a bad name. To serialize complex objects to and from a text format and to make markup that is easily parsed and generated are the real strengths of XML. The best used XML is XML nobody ever realizes is even there. Certainly that is not true completely with FreeSWITCH today but as we scale and more GUI's and config apps are made, the curve will be in our favor and most people besides only the truly advanced users will ever have to see the XML again. --- Anthony Minessale 全文见：http://www.freeswitch.org/node/123。

```
|      ├── default
|      ├── public
|      └── skinny-patterns
├── directory
|      └── default
├── ivr_menus
├── jingle_profiles
├── lang
|      ├── en
|      |      ├── demo
|      |      ├── dir
|      |      ├── ivr
|      |      └── vm
|      ├── fr
|      |      ├── demo
|      |      ├── dir
|      |      └── vm
├── mrcp_profiles
├── sip_profiles
|      ├── external
|      └── internal
└── skinny_profiles
```

其中最重要的一个配置文件是 freeswitch.xml，它的作用是将所有配置文件“粘”到一起。只要有一点点 XML[⊖]基础知识，都可很容易理解这些配置。其中，X-PRE-PROCESS 标签是 FreeSWITCH 中特有的，它称为预处理指令，用于设置一些变量和装入其他配置文件。

在 XML 加载阶段，FreeSWITCH 的 XML 解析器会先将预处理命令展开，在 FreeSWITCH 内部生成一个大的 XML 文档。log/freeswitch.xml.fsxml 是 FreeSWITCH 内部 XML 的一个内存镜像（注意，它在 log 目录中，而不是在 conf 目录中，由于它是动态生成的，所以用户不应该手工编辑它）。它对调试非常有用：假设你不慎弄错了某个标签，又不知道它错在哪了，可以尝试让 FreeSWITCH 重新加载 XML（reloadxml），这时在 FreeSWITCH 的日志中就可以看到 XML 某一行出错的提示，在 freeswitch.xml.fsxml 就能很容易地定位到这一行。

完整的 XML 文档分为以下几个重要的部分：configuration（配置）、dialplan（拨号计划）、chatplan（聊天计划）、directory（用户目录）及 phrase（分词），每一部分又有不同的配置。由于完整的 XML 非常大，而且大的 XML 文档已经对 X-PRE-PROCESS 标签进行处理，隐藏了很多配置的细节，因此这里我们还是按单个的文件或目录进行介绍。

5.3.1 freeswitch.xml

freeswitch.xml 是所有 XML 文件的黏合剂，我们先从它讲起。为了便于讲解，下面是

⊖ XML 由标签（Tag）和属性构成。<tag> 和 </tag> 组成一对标签，如果该标签有相关属性，则以 <tag attr="value"></tag> 形式指定。有些标签无须配对，但必须以“ />”关闭该标签定义，如 <other_tag attr="value"/>。

一个精简过的 freeswitch.xml。

```
<?xml version="1.0"?>
<document type="freeswitch/xml">
    <!-- #comment 这是一个配置文件，本行是注释 -->

    <X-PRE-PROCESS cmd="include" data="vars.xml"/>

    <section name="configuration" description="Various Configuration">
        <X-PRE-PROCESS cmd="include" data="autoload_configs/*.xml"/>
    </section>
</document>
```

众所周知，XML 文档是一种树型的结构。在这里，它的根是 document，在 document 中，有许多 Section，每个 Section 都对应一部分功能。其中有两个 X-PRE-PROCESS 预处理指令，它们的作用是将 data 参数指定的文件内容包含（include）到当前文件中来。

上述 freeSwitch.xml 文件没有特别的作用，它就是将不同的配置文件“包含”到不同的部分（Section）中，从而生成一个大的 XML 配置文件。最后大的配置文件将类似下列代码所示：

```
<?xml version="1.0"?>
<document type="freeswitch/xml">

    <section name="configuration" description="Various Configuration">
        <configuration name="mod1.conf" description="Mod 1"/>
        <configuration name="mod2.conf" description="Mod 2"/>
        <configuration name="mod3.conf" description="Mod 3"/>
    </section>
    <section name="dialplan" description="XML Dialplan ">
        <context name="default " description="default Dialplan"/>
        <context name="public" description="public Dialplan"/>
    </section>
</document>
```

可以看出，configuration 这个 Section 中有很多 configuration，而 dialplan 这个 Section 中有很多 Context。关于这些配置项的具体含义我们将在后面相关的章节讲解，在此就不多讲了。

由于 X-PRE-PROCESS 是一个预处理指令，FreeSWITCH 在加载阶段只对其进行简单替换，并不进行语法分析，因此对它进行注释是没有效果的，这是一个新手常犯的错误。下面举例说明一下。

假设 vars.xml 的内容如下，它是一个合法的 XML：

```
<!-- this is vars.xml -->
<vars>
    <var1>xxxxx</var1>
    <var2>xxxxx</var2>
</vars>
<!-- end of vars.xml -->
```

若在调试阶段想把一条 X-PRE-PROCESS 指令注释掉：

```
<!-- <X-PRE-PROCESS cmd="include" data="vars.xml"/> -->
```

当 FreeSWITCH 预处理时，还没有到达 XML 解析阶段，也就是说它还不认识 XML 注释语法，而仅会机械地将预处理指令替换为如下的 vars.xml 里的内容：

```
<!--    <!-- this is vars.xml -->
<vars>
    <var1>xxxxx</var1>
    <var2>xxxxx</var2>
</vars>
<!-- end of vars.xml --> -->
```

因而，由于 XML 的注释不能嵌套，这里便产生了错误的 XML。解决办法是破坏掉 X-PRE-PROCESS 的定义，如作者常用下面两种方法：

```
<xX-PRE-PROCESS cmd="include" data="vars.xml"/>
<XPRE-PROCESS cmd="include" data="vars.xml"/>
```

由于 FreeSWITCH 不认识 xX-PRE-PROCESS 及 XPRE-PROCESS，因此它会忽略该行，相当于注释掉了。

5.3.2 vars.xml

vars.xml 主要通过 X-PRE-PROCESS 指令定义了一些全局变量，如：

```
<X-PRE-PROCESS cmd="set" data="domain=$${local_ip_v4}"/>
<X-PRE-PROCESS cmd="set" data="domain_name=$${domain}"/>
<X-PRE-PROCESS cmd="set" data="hold_music=local_stream://moh"/>
<X-PRE-PROCESS cmd="set" data="use_profile=internal"/>
```

读者可能已经注意到了，该指令使用 set 表示要设置一个变量。如上述第 1 行，将 domain 这一变量的值设置为 $${local_ip_v4}。那么 $${local_ip_v4} 是什么意思呢？它又是从哪里来的呢？

可以这样讲，在这里使用 X-PRE-PROCESS 设置的变量都称为全局变量，它们在 FreeSWITCH 运行期间永远都是有效的。而后面可能还会遇到局部变量，它们通常在拨号计划中，在一个呼叫的生命周期中才有效。如果需要引用这些变量，则全局变量以 $${var} 表示，临时变量以 ${var} 表示。

在加载 vars.xml 之前，FreeSWITCH 就已经“算”出并设置了一些全局变量，也就是说有些变量是系统在运行时自动设置的，其有默认的值，如表 5-2 所示。

表 5-2 系统自动设置的变量

变 量	说 明
hostname	主机名
local_ip_v4	本地 IP（IPv4）
local_mask_v4	本地掩码（IPv4）
local_ip_v6	本地 IP（IPv6）
switch_serial	随机产生的序列号

（续）

变　量	说　明
base_dir	FreeSWITCH 的安装路径
recordings_dir	录音路径，可以在启动时用 –recordings 参数指定
sound_prefix	声音文件的路径前缀，默认是 /usr/local/freeswitch/sounds/en/us/callie
sounds_dir	声音文件的路径，默认是 /usr/lcoal/freeswitch/sounds
core_uuid	自动生成的 uuid 字符串，用于标志本 FreeSWITCH 实例，每次启动都会改变
zrtp_enabled	是否启用 zrtp
nat_public_addr	检测到的 NAT 公网地址
nat_private_addr	NAT 私网地址
nat_type	NAT 类型

在实际使用中，可以使用 global_getvar 或这个 API 命令来查看这些变量的值，如：

```
freeswitch> global_getvar sound_prefix
/usr/local/freeswitch/sounds/en/us/callie
freeswitch > global_getvar local_ip_v4
192.168.7.6
```

由于这些变量是在 vars.xml 加载前设置的，因而可以在 varx.xml 中覆盖它们，如：

```
<X-PRE-PROCESS cmd="set" data="local_ip_v4=192.168.1.123"/>
```

上面的设置经常会用到，因为有时候 FreeSWITCH 自动“算”出的值可能不是你想要的，如上面的 local_ip_v4 的值，在服务器有多个网卡的情况下，可能希望它能得到另外一个网卡的 IP 地址，这时候就可以通过手动的方式设置该变量的 IP 来实现。

5.3.3 autoload_configs 目录

autoload_configs 目录下的各种配置文件会在系统启动时装入。一般来说都是模块级的配置文件，每个模块对应一个（注意，并不是所有的模块都有配置文件）。文件名一般以“模块名 .conf.xml”的方式命名（模块名中不包含“mod_”，如 sofia.conf.xml）。

各模块的配置文件不尽相同，其中的配置项也都是由不同的模块分别解析的。这里我们以 sofia.conf.xml 为例简单介绍一下。

```
<configuration name="sofia.conf" description="sofia Endpoint">

  <global_settings>
    <param name="log-level" value="0"/>
    <!-- <param name="auto-restart" value="false"/> -->
    <param name="debug-presence" value="0"/>
    <!-- <param name="capture-server" value="udp:homer.domain.com:5060"/> -->
  </global_settings>

  <!--
      The rabbit hole goes deep.  This includes all the
      profiles in the sip_profiles directory that is up
      one level from this directory.
```

```
  -->
  <profiles>
      <X-PRE-PROCESS cmd="include" data="../sip_profiles/*.xml"/>
  </profiles>

</configuration>
```

该配置文件比较简单，首先它定义了一个 configuration 标签，其中 name 表示名字，这里是 sofia.conf，而后面的 description 是一个简单的描述。

其实，sofia.conf.xml 的文件名并不重要，完全可以改成其他的名字，只要扩展名是 .xml 就可以正常被 freeswitch.xml 中的 X-PRE-PROCESS 预处理指令装入。这里重要的是，这个配置文件中的 configuration 标签的 name 属性，mod_sofia 在启动时会向 XML 注册表中查找 name 为“sofia.conf”的 configuration，进而访问其下面的配置参数。

global_settings 标签定义了一些全局参数，具体参数的意义我们在此就不多解释了。

profiles 标签可通过 X-PRE-PROCESS 指令装入其他的配置文件，这些配置文件一般都是每个文件描述了一个 profile（见默认的 conf/sip_profiles/internal.xml），关于这些 profile 我们将在第 8 章中详细讲解。在此我们仅看一下 profile 装入后的效果，具体如下：

```
<profiles>
        <profile name="profile1"></profile>
        <profile name="profile2"></profile>
</profiles>
```

另外，autoload_configs 目录中有一个特殊的 modules.conf.xml，其决定了 FreeSWITCH 启动时自动加载哪些模块。如下面的配置片断，如果需要在 FreeSWITCH 启动时自动加载某个模块，就在这里添加一行，如果不需要，就注释掉或直接删除。下面列出了该配置文件的前几行：

```
<configuration name="modules.conf" description="Modules">
  <modules>

    <!-- Loggers (I'd load these first) -->
    <load module="mod_console"/>
    <load module="mod_logfile"/>
<!-- <load module="mod_syslog"/> -->
```

最后，autoload_configs 目录中还有一个 post_load_modules.conf.xml 文件，其格式和用法与 modules.conf.xml 差不多，不同的是其中定义的模块加载时间比较晚。

5.3.4 其他

上面我们大体了解了配置文件的目录结构和简单的配置格式。其他配置文件的配置方法也大同小异。至于具体的配置项的含义只有到了具体的应用时才有意义，因此，我们后面用到某些项时，会详细进行说明。不过，在结束本节之前，我们还需简单介绍以下几个目录，因为我们在后面的章节中可能会经常提到它们：

❑ dialplan 目录：该目录下的文件定义了 XML 拨号计划（Dialplan）。拨号计划是 Free-

SWITCH 中很重要的一部分，它用于对进电话进行路由。我们将会在第 6 章专门讲解拨号计划。

- ivr_menues 目录：该目录下存放了默认的一些 IVR 菜单的例子。
- directory 目录：该目录中的配置文件决定了当 FreeSWITCH 作为注册服务器时，哪些用户可以注册，即用于配置本地用户，其中的配置信息称为用户目录。FreeSWITCH 的用户目录支持多个域（Domain），每个域可以写到一个 XML 文件中。默认的配置包括一个 default.xml，其中定义了 1000 ~ 1019 一共 20 个用户（实际上是装入了下一级子目录中的 20 个配置文件）。

5.4　XML 用户目录

我们在 5.3.4 节提到 XML 用户目录决定了哪些用户可以注册到 FreeSWITCH 上。在此，我们再深入分析一下。

SIP 并不要求一定要注册才可以打电话，但是通话前的用户认证参数仍需要在用户目录中进行配置。

用户目录的默认配置文件在 conf/directory/ 下，系统自带的配置文件为 default.xml，其实现代码如下：

```
<domain name="$${domain}">
  <params>
    <param name="dial-string"
        value="{^^:sip_invite_domain=${dialed_domain}:presence_id=${dialed_
user}@${dialed_domain}}${sofia_contact(*/${dialed_user}@${dialed_domain})}"/>
  </params>

  <variables>
    <variable name="record_stereo" value="true"/>
    <variable name="default_gateway" value="$${default_provider}"/>
    <variable name="default_areacode" value="$${default_areacode}"/>
    <variable name="transfer_fallback_extension" value="operator"/>
  </variables>

</domain>
```

该配置文件决定了哪些用户能注册到 FreeSWITCH 中。一般来说，所有用户都应该属于同一个 domain（除非想使用多个 domain，后面我们会有例子）。这里的 $${domain} 这个全局变量是在 vars.xml 中设置的，它默认是主机的 IP 地址，可以修改为使用一个域名。params 中定义了该 Domain 中所有用户的公共参数。在这里只定义了一个 dial-string，这是一个至关重要的参数。在使用 user/username 或 sofia/internal/username@domain 这样的呼叫字符串时，FreeSWITCH 会根据 username（以及 domain）找到该 dial-string，并最终扩展成用户实际的 SIP 地址[⊖]。其中 sofia_contact 是一个 API 命令，它会根据用户的注册地址扩展

⊖ 这个呼叫字符串比较复杂，需要许多知识才能理解它，因此我们留到本书的第二部分来详细研究。

成相应的呼叫字符串。

variables 则定义了一些公共变量，在用户主叫或被叫时，这些变量会绑定到相应的 Channel 上形成 Channel Variable。

在 domain 中还定义了许多组（group），组里面包含很多用户（user）。

```
<groups>
  <group name="default">
    <users>
      <X-PRE-PROCESS cmd="include" data="default/*.xml"/>
    </users>
  </group>
</groups>
```

在这里，组名 default 并没有什么特殊的意义，它只是随便起的，可以将其修改成任何值。在用户标签中，使用预处理指令装入了 default 目录中的所有 XML 文件。可以看到，在 default 目录中，每个用户都对应一个文件。

也可以定义其他的用户组，组中的用户并不需要是完整的 XML 节点，也可以是一个指向已存在用户的“指针”，如下代码就是使用 type="pointer" 定义的指针。

```
<group name="sales">
  <users>
    <user id="1000" type="pointer"/>
    <user id="1001" type="pointer"/>
    <user id="1002" type="pointer"/>
  </users>
</group>
```

虽然这里设置了组，但使用组并不是必需的。如果不打算使用组，将用户节点（users）直接放到 domain 的下一级也是可以的。但使用组可以比较方便地支持像群呼、代接之类的业务。例如，使用 group_call 可以同时或顺序呼叫某个组的用户。

其实与用户相关的设置很直观，下面显示了 Alice 这个用户的设置：

```
<user id="alice">
  <params>
    <param name="password" value="$${default_password}"/>
    <param name="vm-password" value="alice"/>
  </params>
  <variables>
    <variable name="toll_allow" value="domestic,international,local"/>
    <variable name="accountcode" value="alice"/>
    <variable name="user_context" value="default"/>
    <variable name="effective_caller_id_name" value="Alice"/>
    <variable name="effective_caller_id_number" value="1000"/>
    <variable name="outbound_caller_id_name" value="$${outbound_caller_name}"/>
    <variable name="outbound_caller_id_number" value="$${outbound_caller_id}"/>
    <variable name="callgroup" value="techsupport"/>
  </variables>
</user>
```

由上面代码可知，实际上 params 和 variables 可以出现在 user 节点中，也可以出现在 group 或 domain 中。当它们有重复时，优先级顺序由高到低依次为 user、group、domain。

当然，用户目录还有一些更高级的设置，我们留待以后研究。

5.5 呼叫相关概念

本节我们再来复习一下第 3 章的内容，FreeSWITCH 是一个 B2BUA，我们还是以图 3-2 为例。

如果 Bob 与 Alice 通话，典型的呼叫流程主要有以下两种：

- Bob 向 FreeSWITCH 发起呼叫，FreeSWITCH 接着启动另一个 UA 呼叫 Alice，两者通话。
- FreeSWITCH 同时呼叫 Bob 和 Alice，两者接电话后 FreeSWITCH 将 a-leg 和 b-leg 桥接（bridge）到一起，两者通话。

其中第二种流程还有一种变种。例如，市场上有人利用上、下行通话的不对称性卖电话回拨卡获取利润。比方说，他们可以把 FreeSWITCH 作为服务器，把回拨卡卖给 Bob。Bob 按回拨卡上的号码呼叫 FreeSWITCH，FreeSWITCH 不应答，而是在获得 Bob 的主叫号码后直接挂机。然后 FreeSWITCH 回拨 Bob，Bob 接听后 FreeSWITCH 启动一个 IVR 程序指示 Bob 输入 Alice 的号码。最后 FreeSWITCH 再呼叫 Alice ⊖。

在实际应用中，由于涉及回铃音、呼叫失败等，情况可能要复杂得多。在此我们就不深入讨论了，而是针对简单的呼叫流程介绍一下与呼叫相关的基本概念。

5.5.1 来去话、Session 、Channel 与 Call

在前面所说的第一种呼叫流程中，Bob 到 FreeSWITCH 的通话称为来话（注意，在这里我们都是针对 FreeSWITCH 来说的），而 FreeSWITCH 作为一个 B2BUA 再去呼叫 Alice 时，就称为去话。显而易见，在第二种呼叫流程中两路通话（两个 leg）都是去话。

无论来话还是去话，对每一次呼叫，FreeSWITCH 都会启动一个 Session（会话，它包含 SIP 会话，SIP 会在每对 UAC-UAS 之间生成一个 SIP Session），用于控制整个呼叫，它会一直持续到通话结束。其中，每个 Session 都控制着一个 Channel（通道，又称信道），Channel 是一对 UA 间通信的实体，相当于 FreeSWITCH 的一条腿（leg），每个 Channel 都用一个唯一的 UUID 来标识，称为 Channel UUID。另外，Channel 上可以绑定一些呼叫参数，称为 Channel Variable（通道变量）。Channel 中可能包含媒体（音频或视频流），也可能不包含。通话时，FreeSWITCH 的作用是将两个 Channel（a-leg 和 b-leg，通常先创建的或占主动的叫 a-leg）桥接（bridge）到一起，使双方可以通话。这两路桥接的通话（两条腿）在逻辑上组成一个通话，称为一个 Call。

在通话中，媒体（音频或视频）数据流在 RTP 包中传送。一般来说，Channel 是双向的，

⊖ 在整个过程中，Bob 和 Alice 的电话都不会被扣费，费用将在回拨卡账号上扣除，通常会比 Bob 直接呼叫 Alice 便宜。这种方法比较复杂，用户体验也不好，故我们不赞成这种做法。在此只是举一个例子，说明 FreeSWITCH 可以这样用。

因此媒体流会有发送（Send/Write）和接收（Receive/Read）两个方向。

5.5.2 回铃音与 Early Media

为了便于说明，我们假定 A 与 B 不在同一台交换机（服务器）上（如在 PSTN 通话中可能不在同一座城市），中间需要经过两台交换机中转，如图 5-2 所示。

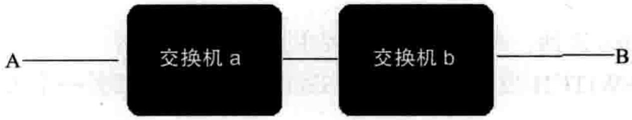


图 5-2 A 与 B 通过两台交换机通话

假设图 5-2 所示是在 PSTN 网络中，A 呼叫 B，B 话机开始振铃，A 端可听到回铃音（Ring Back Tone）。在早期，B 端所在的交换机 b 只向 A 端交换机 a 传送地址全（ACM）信号，证明呼叫是可以到达 B 的，A 端听到的回铃音的铃流是由 A 端所在的交换机生成并发送的。但后来，为了在 A 端能听到 B 端特殊的回铃音（如“您拨打的电话正在通话中…”或“对方暂时不方便接听您的电话”尤其是现代交换机，支持各种个性化的彩铃[⊖]），回铃音只能由 B 端交换机发送。在 B 接听电话前，回铃音和彩铃是不收费的（不收取主叫 A 的本次通话费。彩铃费用一般是在被叫端即 B 端以月租或套餐形式收取的）。这些回铃音就称为 Early Media（早期媒体）。在 SIP 通信中，它是由 SIP 的 183（带有 SDP）消息描述的。

理论上讲，B 接听电话后交换机 b 可以一直不向交换机 a 发送应答消息，而是将真正的话音数据伪装成 Early Media，以实现“免费通话”。但这种应用是有限制的，大多数交换机允许 Early Media 的时间不会太长，如 1 分钟，以防止不守规则的人进行免费通话。

5.5.3 全局变量与局部变量

在 5.3.2 节我们讲到，可以使用 X-PRE-PROCESS 在 FreeSWITCH 中设置一些变量（包括自动生成的变量），在后续使用时可以用 $${var} 的形式来进行引用。这些变量是全局有效的，因而称为全局变量。另外一些变量是在 Dialplan、Application 或 Directory 中设置的，它们会影响呼叫流程且可以被动态改变。这些变量一般与一个呼叫有关，严格地说是与一个 Channel 有关，因而又称为 Channel Variable，即通道变量。通道变量可以以 ${var} 的形式引用。全局变量仅在预处理阶段（系统启动时或重新装载 - reloadxml 时）被求值，一般用于设置一些系统一旦启动就不会轻易改变的量，如 $${domain} 或 $${local_ip_v4} 等。而局部变量（即通道变量）仅在 Channel 的生命周期中有效。所以，两者最大的区别是，$${var} 只在加载时求值一次[⊜]，而 ${var} 则在每次执行时都求值（如一个新电话进来时）。

⊖ 彩铃称为 CRBT，即 Color Ring Back Tone。

⊜ 如果更改 XML 配置文件中的全局变量，一般需要重新启动才能重新加载。在 UNIX 类系统上，也可以向 FreeSWITCH 发送 HUP 信号让它重新加载和解析全局的配置文件，实现命令是 kill -HUP <FreeSWITCH 的 pid>。

在呼叫过程中，某些变量可以改变 Channel 的行为。另外，也可以使用自定义的通道变量来存储随路数据等。

在实际使用中会发现，有些变量在显示时（可以使用 dp_tools 中的 info App 显示，后面会讲到）是以 variable_ 开头的，但在实际引用时要去掉这些开头的 variable_。如 variable_username，引用时要使用 ${username}。我们将在下一章深入讨论通道变量。

5.6 小结

FreeSWITCH 是由一个稳定的核心及外围的可加载模块组成的。核心代码经过精心设计及严格测试，保证了它的稳定性。外围的模块则保证了它的可扩展性。核心与模块间通过 Public API 通信，而模块间通过事件系统（Event）通信，事件机制消除了模块间的耦合，进一步增加了其可伸缩性。

与第 4 章类似，本章也介绍了一些基本概念，如 Session、Channel、Channel Variable 等，这里介绍的基本概念后面会多次涉及，读者在后面遇到时也不妨回来复习一下，以加深对它们的理解。

第 6 章

拨号计划

拨号计划（Dialplan）是 FreeSWITCH 中至关重要的一部分。它的主要作用就是对电话进行路由（从这一点上来说，相当于一个路由表），决定和影响通话的流程。说得简明一点，就是当一个用户拨号时，对用户所拨的号码进行分析，进而决定下一步该做什么。当然，实际上它所能做的比你想象的要强大得多。

我们在第 3 章中已经介绍过了如何修改拨号计划，单从配置文件看，拨号计划还算比较简单。实际上，它的概念也不是很复杂。如果你理解正则表达式，那你应该能看懂系统自带的大部分的配置。但是在实际应用中，有许多问题还是常常令初学者感到疑惑。主要的问题是，要理解 Dialplan，还需要了解 FreeSWITCH 是怎样工作的、API 与 App 的区别、呼叫字符串等。这些概念我们在第 4 章和第 5 章都讲过，如果读者在学习本章的过程中有什么疑问，不妨再倒回去复习一下。

Dialplan 是 FreeSWITCH 中一个抽象的部分，它可以支持多种不同的格式，如 Asterisk 风格的格式（由 mod_dialplan_asterisk 提供）、LUA、inline 等。在实际使用中，用得最多的还是 XML 格式。下面，我们就从 XML Dialplan 开始，学习一下 Dialplan 的基本概念和使用方法。

6.1 XML Dialplan

XML Dialplan 由一系列的 XML 配置文件组成，这些 XML 可以是静态配置的，也可以使用动态配置方式从其他服务器或脚本中动态获取。Dialplan 有特定的结构，FreeSWITCH 通过解析相关的结构，可对 Dialplan 进行路由的呼叫，决定执行何种动作或流程。

6.1.1 配置文件的结构

拨号计划的配置文件默认在 conf/dialplan 目录中，我们在第 5 章中讲过，它们是在 freeswitch.xml 中定义的，是由以下预处理指令装入的：

```
<X-PRE-PROCESS cmd="include" data="dialplan/*.xml"/>
```

拨号计划由多个 Context[⊖]组成。每个 Context 中有多个 Extension[⊜]。所以 Context 就是多个 Extension 的逻辑集合，它相当于一个分组。一个 Context 中的 Extension 与其他 Context 中的 Extension 在逻辑上是隔离的。

下面是 Dialplan 的完整结构：

```
<?xml version="1.0"?>
<document type="freeswitch/xml">
    <section name="dialplan" description="Regex/XML Dialplan">
        <context name="default">
            <extension name="Test Extension">
            </extension>
        </context>
    </section>
</document>
```

Extension 相当于路由表中的表项，其中，每一个 Extension 都有一个 name 属性，name 可以是任意合法的字符串，本身对呼叫流程没有任何影响。但给它取一个好听的名字，有助于你在查看 Log 时发现它。

在 Extension 中可以对一些 condition（测试条件）进行判断，如果满足测试条件所指定的表达式，则执行对应的 Action（动作）。

例如，我们将下列 Extension 配置加入到 conf/dialplan/default.xml 中，并作为第一个 Extension：

```
<extension name="My Echo Test">
  <condition field="destination_number" expression="^echo|1234$">
    <action application="answer" data=""/>
    <action application="echo" data=""/>
  </condition>
</extension>
```

FreeSWITCH 安装时，提供了很多例子，为了避免与提供的例子冲突，强烈建议初学者在学习时把自己写的 Extension 放在最前面[⊜]。当然笔者这里所说的最前面并不是 default.xml 的第一行，而是紧跟着 Context，放到第一个 Extension 的位置，就是以下语句的后面（你通常能在第 13 ～ 14 行找到它们）：

```
<include>
    <context name="default">
```

用你喜欢的文本编辑器编辑好并存盘后，在 FreeSWITCH 控制台上执行 reloadxml 或

⊖ 可以直译为上下文、环境、呼叫源等，但都感觉词不达意，因此在本书中我们直接称为 Context。如果把整个 Dialplan 看作一张路由表的话，Context 就相当于路由表项中不同的逻辑分组或部分。另外，这里的 Context 以及下文讲的 Extension、Action 等名词，虽然有的可以翻译成中文，但都不能准确的反映原文的意思。因此，我们这里以专有名词对待，统一使用英文，并以首字母大写形式呈现。而在实际的配置文件中，则按惯例以小写表示。

⊜ 分支，在简单的 PBX 中也可以认为是分机号，但很显然，Extension 涵盖的内容远比分机号多。

⊜ 如果你不是初学者，则不建议放到最前面，至少要放到 unloop 那个 Extension 后面。

按 F6 键，使 FreeSWITCH 重新读入你修改过的配置文件。并按 F8 键（相当于命令 console loglevel debug）将 Log 级别设置为 Debug，以显示详细日志。然后注册软电话，并拨叫 1234 或 echo ⊖。接下来你将会在 FreeSWITCH 控制台上看到很多 Log，注意如下的行：

```
1 Processing Seven <1000>->1234 in context default
2 parsing [default->My Echo Test] continue=false
3 Regex (PASS) [Echo Test] destination_number(1234) =~ /^echo|1234$/ break=on-false
4 Action answer() Action echo()
```

在笔者的终端上，上面的第 1 行是以绿色显示的（此处加粗）。当然，为了排版方便，笔者省去了 Log 中的日期以及其他一些不关键的信息。下面我们对上面的代码进行简单分析：

- 第 1 行，Processing 说明是在处理 Dialplan，其中 Seven 是笔者客户端的 SIP 名字，1000 是分机号（即主叫号码），1234 是拨叫的号码（被叫号码），这里笔者直接拨叫了 1234。它完整意思是说：呼叫已经达到路由阶段，要从 XML Dialplan 中查找路由，该呼叫来自 Seven，分机号是 1000，它所呼叫的被叫号码是 1234（或 echo，如果你拨的是 echo 的话）。
- 第 2 行，呼叫进入 parsing(解析 XML) 阶段，它首先根据呼叫的来源找到 XML 中的一个 Context，此处是 default ⊜。它找到的第一个 Extension 的 name 是 My Echo Test（还记得吧？我们把它放到了 Dialplan 的最前面）。continue=false 的意义我们稍后再讲。
- 第 3 行，由于该 Extension 中有一个 Condition，它的测试条件是 destination_number，也就是被叫号码，所以 FreeSWITCH 测试被叫号码（这里是 1234）是否与配置文件中的正则表达式相匹配。^echo|1234$ 是正则表达式，它匹配字符串 echo 或 1234。所以这里匹配成功，Log 中显示 Regex(PASS)。匹配成功后，电话会进入执行阶段（EXECUTE），在 Log 中会看到如下信息：

```
(sofia/internal/1000@192.168.7.7) State Change CS_ROUTING -> CS_EXECUTE
```

第一个要执行的动作是 answer，它是一个 App，用于对来话进行应答（由于这是一个 SIP 呼叫，因而底层的信令会给主叫发送 200 OK 消息），然后执行 echo（它也是一个 App，在第 4 章我们讲过，这些 App 大部分来自于 mod_dptools），它的作用就相当于一个回音壁，你说什么都反原封不动地反弹回来，所以你能听到自己的声音。

实际上，在这个例子中，我们呼叫 1234 时创建了一个单腿的呼叫，与其说我们跟 FreeSWITCH 在通话，还不如说我们在跟 FreeSWITCH 中的一个 App 在通话。而 XML Dialplan 只是帮助我们找到这个（些）App。

上面所说的是最简单的路由查找，为了简单起见，我们自己写了一个例子。实际上，系统自带的一些 Dialplan 的例子很有代表性。也许在第 3 章你已经测试过一些了，但我相信当

⊖ 大部分软电话都能呼叫字母，如 Zoiper，XLite 可以使用空格键在数字和字母间切换。能以字母作为被叫号码也是软电话与传统硬电话最大的不同。

⊜ 它是在用户目录中定义的，参见第 5 章。用户目录配置文件中有一项 <variable name="user_context" value="default"/>，说明，如果 1000 这个用户发起呼叫，则它的 context 就是 default，所以，待呼叫进行到路由阶段后，要从 XML Dialplan 中的 default 这个 Context 查起。

时你只是测的功能，而没有仔细看 Dialplan 的查找过程。下面我们一起来试一下系统自带的 echo 的例子。这次我们呼叫 9196（参见表 2-1）。在产生的 Log 中，我们还是从“绿色的行”（Processing）开始看：

```
 1 Processing Seven <1000>->9196 in context default
 2 parsing [default->My Echo Test] continue=false
 3 Regex (FAIL) [Echo Test] destination_number(9196) =~ /^echo|1234$/ break=on-false
 4 parsing [default->unloop] continue=false
 5 Regex (PASS) [unloop] ${unroll_loops}(true) =~ /^true$/ break=on-false
 6 Regex (FAIL) [unloop] ${sip_looped_call}() =~ /^true$/ break=on-false
 7 parsing [default->tod_example] continue=true
此处省略很多行 ......
42 Regex (FAIL) [video_playback] destination_number(9196) =~ /^9194$/ break=on-false
43 parsing [default->delay_echo] continue=false
44 Regex (FAIL) [delay_echo] destination_number(9196) =~ /^9195$/ break=on-false
45 parsing [default->echo] continue=false
46 Regex (PASS) [echo] destination_number(9196) =~ /^9196$/ break=on-false
47 Action answer()
48 Action echo()
```

由于默认的配置比较长，因而产生的 Log 比较多。在此，我们先不关注匹配失败的部分（FAIL 的行，如第 3 行和第 44 行）。可以看到，除第 5 行外⊖，前面的正则表达式匹配大多数都没有成功，只是到最后匹配到 ^9196$ 才成功（第 46 行，注意其中的 PASS），成功后先执行 answer（应答），然后执行 echo（回声）。

在这一节里，我们之所以花了很多篇幅来讲解如此简单的问题，是因为笔者是想让读者明白，这一节最重要的不是讲 Dialplan，而是要告诉读者如何看 Log。永远记住：**遇到 Dialplan 的问题，按 F8 键打开详细的 Debug 级别的日志，尝试打一个电话，并从 Log 中绿色的行开始看起**⊜。我们的第一个例子虽然只有短短的四行 Log，但是它包含了所有你需要的信息。

6.1.2 默认的配置文件简介

系统默认提供的配置文件包含三个 Context，分别是 default、features 和 public。它们分别在三个 XML 文件中。default 是默认的 Dialplan，一般来说注册用户都可以通过它来打电话，如拨打其他分机或外部电话等。而 public 一般用于接收外来呼叫，因为从外部进来的呼叫是不可信的，所以要进行更严格的控制。这个应该很容易理解，比方说：你肯定不想让外部进来的、不可信的电话通过你的网关进行国内或国际长途呼叫，并为这些呼叫买单。

当然，这么说不是绝对的，等你熟悉了 Dialplan 的概念之后，可以发挥你的想象力进行任意有创意的配置。

⊖ 第 5 行是一个组合的 Condition，虽然第一个 Condition 被 PASS 了，但第二个 FAIL 了，因此总的来说还是 FAIL。关于 Condition 的详细介绍，见 6.1.5 节。

⊜ 当然，这里所说的绿色的行纯属巧合，实际上应该是解析 Diaplan 的那一行，即从“Processing ...”这一行开始看，它是 INFO 级别的日志，在一大堆黄色的行中比较显眼。当然，如果你的终端不支持彩色，那就需要多费一下眼睛了。

在 default 和 public 中，又通过 include 预处理指令分别加入了 default 和 include 目录中的所有 .xml 文件。这些目录中的文件仅包含一些额外的 Extension。由于 Dialplan 在处理的时候是一定的顺序进行的，所以一定要注意这些文件的装入顺序。通常这些文件都是按文件名排序的，如 00_、01_ 等等。如果你想添加新的 Extension，可以在这些目录里创建文件。但要注意，这些文件的优先级一般比直接写在 default.xml 或 public、xml 中要低（取决于 include 语句的位置）。前面已经说过，若你还不熟悉系统提供的默认的 Dialplan，则有可能因不当操作致使出现与既有的 Dialplan 相冲突的情况。当然，通过前面的章节我们已经学会如何查看 Log，所以即使出现问题也应该能很容易找到问题所在。但在本书中，笔者还是坚持将新加的 Extension 加在 Dialplan 中的最前面，以节省篇幅，并便于说明问题。

实际上，由于在处理 Dialplan 时要对每一项进行正则表达式匹配，这是非常影响效率的。所以在生产环境中，往往要删除这些默认的 Dialplan，只保留或添加有用的部分。但是这里我们还不能删，因为里面有很多例子我们还没有好好学习呢。

在深入讲解这些配置文件之前，我们先来看几个基本的概念，然后再在本章后面的部分讲解几个实际的例子。

6.1.3 正则表达式⊖

Dialplan 使用与 Perl 兼容的正则表达式（Perl-Compatible Regular Expressions，PCRE）匹配算法。熟悉编程的读者肯定已经很熟悉它了，为了方便不熟悉的读者，在这里我们通过几个小例子来简单介绍一下，如表 6-1 所示。

表 6-1 几个正则表达式的例子及说明

例 子	说 明
^1234$	“^”匹配字符串的开头，“$”匹配字符串的结尾，所以本表达式严格匹配 1234
^1234|5678$	“|”是或的意思，表示匹配 1234 或 5678
^123[0-9]$	“[]”表示匹配其中的任意一个字符，其中的“-”表示一个区间，即 0 到 9，它等于 [0123456789]，也就是说它会匹配 1230，1231，1232，…，1239
^123\d$	同上，\d 等于 [0-9]
^123\d+$	“+”号表示匹配 1 个或多个它前面的字符，这里，由于“+”前面是“\d”，所以它就等于 1 个或多个数字，实际上它匹配任意以 123 开头的至少 4 位数的数字串，如 1230、12300、12311、123456789 等
^123\d*$	“*”号与“+”号的不同在于，它匹配 0 个或多个前面的字符。所以，它匹配以 123 开头的至少 3 位数的数字串，如 123，123789
^123	跟上面一样，由于没有结尾的 $，它匹配任何以 123 开头的数字串，但除此之外，它还匹配后面是字母的情况，如 123abc
123$	匹配任何以 123 结尾的字符串
^123\d{5}$	{5} 表示精确匹配 5 位，包含它前面的一个字符。在这里，它匹配以 123 开头的所有 8 位的电话号码

⊖ 参考资料：http://zh.wikipedia.org/zh/ 正则表达式和 http://wiki.freeswitch.org/wiki/Regular_Expression。

(续)

例子	说明
^123(\d+)$	“()”在匹配中不起作用，因而此处跟 ^123\d+ 是相同的；但它对匹配结果有作用，匹配结果中除 123 之外的数字都将存储在 $1 这个变量中，在后面可以引用
^123(\d)(\d+)$	如果用它跟 12345678 匹配，则匹配成功，结果是 $1=4，$2=5678
.	最后说明，“.”匹配任意一个字符，如果你写了“.*”，则它会匹配任意字符串

FreeSWITCH 提供了简单的 API 可以测试你写的正则表达式是否正确，只需要在命令行上输入“regex 要匹配的字符串 | 正则表达式”即可，如（返回 true 表示匹配）：

```
freeswitch> regex 1234 | \d
true
freeswitch> regex 1234 | \d{4}
true
freeswitch> regex 1234 | \d{5}
false
freeswitch> regex 1234 | ^123
true
```

简单的正则表达式比较容易理解，更深入的学习请查阅相关资料。正则表达式功能很强大，但配置不当也容易出现错误，轻则造成电话不通，重则可能会造成误拨或套拨[⊖]，带来经济损失。

6.1.4 通道变量

在 FreeSWITCH 中，每一次呼叫都由一条或多条“腿”（Call Leg）组成，其中的一条腿又称为一个 Channel（通道），每一个 Channel 都有很多属性，用于标识 Channel 的状态、性能等，这些属性称为 Channel Variable（通道变量），可简写为 Channel Var、Chan Var 或 Var。

通过使用 info 这个 App，可以查看所有的通道变量。我们先修改一下 Dialplan，在 default.xml 中加入下列 Extension：

```
<extension name="Show Channel Variable">
    <condition field="destination_number" expression="^1235|info$">
        <action application="info" data=""/>
    </condition>
</extension>
```

为了复习上一节的内容，这一次我们加入到 My Echo Test 这一 Extension 的后面，存盘，然后在 FreeSWITCH 命令行上执行 reloadxml。从软电话上呼叫 1235 或 info，可以看到有很多 Log 输出。还是从绿色的行（Processing）开始看（为节省篇幅，我们删去了后面的一部分日志）：

⊖ 所谓套拨，就是利用拨号规则的漏洞拨打非常规的号码来达到拨打受限号码和节省话费的目的。一般来说，拨打不同号码的资费是不一样的，如 200 是不计费的，在号码前加 17951 打长途比直接拨打要便宜。如果 Dialplan 设置不当，可能会造成没有 17951 权限的用户拨打 20017951139xxxxxxxx 却能打通，这样系统轻则会自动去掉 200，允许该用户拨打 17951；重则可能直接不计费，这种拨号方式就是套拨。防止这种套拨一般可以通过限制所能输入的号码的最大长度来实现。

```
Processing Seven <1000>->1235 in context default
parsing [default->Echo Test] continue=false
Regex (FAIL) [Echo Test] destination_number(1235) =~ /^echo|1234$/ break=on-false
parsing [default->Show Channel Variable] continue=false
Regex (PASS) [Show Channel Variable] destination_number(1235) =~ /^1235|info$/ break=on-false
Action info()
...
EXECUTE sofia/internal/1000@192.168.7.10 info()
2010-10-23 09:46:31.662281 [INFO] mod_dptools.c:1171 CHANNEL_DATA:
Channel-State: [CS_EXECUTE]
Channel-Call-State: [RINGING]
Channel-Name: [sofia/internal/1000@192.168.7.10]
Unique-ID: [cfea988b-2dc4-42ec-b731-2cd7ea864fc6]
Caller-Direction: [inbound]
Caller-Username: [1000]
Caller-Dialplan: [XML]
Caller-Caller-ID-Name: [Seven]
Caller-Caller-ID-Number: [1000]
Caller-Network-Addr: [192.168.7.10]
Caller-ANI: [1000]
Caller-Destination-Number: [1235]
variable_sip_use_codec_name: [PCMA]
variable_sip_use_codec_rate: [8000]
variable_sip_use_codec_ptime: [20]
variable_read_codec: [PCMA]
variable_read_rate: [8000]
...
```

可以看到，由于我们呼叫的是 1235，它在第三行测试 My Echo Test 中的 1234 的时候失败了，测试会继续往下进行。接下来，测试 1235 的时候成功了，便执行对应的 Action——info 这个 App。它的作用就是把该通道的所有变量都打印到 Log 中。

所有的通道变量都可以在 Dialplan 中访问，使用格式是"${ 变量名 }"，如 ${destination_number}。将下列配置加入 Dialplan 中（存盘，reloadxml 不用再说了吧？）：

```
<extension name="Accessing Channel Variable">
  <condition field="destination_number" expression="^1236(\d+)$">
    <action application="log" data="INFO Hahaha, I know you called ${destination_number}"/>
    <action application="log" data="INFO The Last few digits is $1"/>
    <action application="log" data="ERR This is not actually an error, just jocking :), Have fun!"/>
    <action application="hangup"/>
  </condition>
</extension>
```

这次我们呼叫 1236789，看看结果：

```
1 Processing Seven <1000>->1236789 in context default
2 parsing [default->Echo Test] continue=false
3 Regex (FAIL) [Echo Test] destination_number(1236789) =~ /^echo|1234$/ break=on-false
4 parsing [default->Show Channel Variable] continue=false
5 Regex (FAIL) [Show Channel Variable] destination_number(1236789) =~ /^1235$/ break=on-false
```

```
6 parsing [default->Accessing Channel Variable] continue=false
7 Regex (PASS) [Accessing Channel Variable] destination_number(1236789) =~
/^1236(\d+)$/ break=on-false
8 Action log(INFO Hahaha, I know you called ${destination_number})
9 Action log(NOTICE The Last few digists is 789)
10 Action log(ERR This is not actually an error, just jocking)
11 Action hangup()

12 [DEBUG] switch_core_state_machine.c:157 13 sofia/internal/1000@192.168.7.10
Standard EXECUTE

14 EXECUTE sofia/internal/1000@192.168.7.10 log(INFO Hahaha, I know you called
1236789)
15 [INFO] mod_dptools.c:1152 Hahaha, I know you called 1236789
EXECUTE sofia/internal/1000@192.168.7.10 log(NOTICE The Last few digits is 789)
16 [NOTICE] mod_dptools.c:1152 The Last few digists is 789
EXECUTE sofia/internal/1000@192.168.7.10 log(ERR This is not actually an error,
just jocking :), Have fun!)
17 [ERR] mod_dptools.c:1152 This is not actually an error, just jocking :),
Have fun!
18 EXECUTE sofia/internal/1000@192.168.7.10 hangup()
```

跟前面的例子一样，我们还是从绿色的行开始看。这一次在第 7 行，1236789 匹配了正则表达式 ^1236(\d+)$，并将 789 存储在变量 $1 中。然后在第 8~11 行解析出的 4 个 Action（3 个 log，1 个 hangup）。到此为止，Channel 的状态一直没有变，还处在路由解析的阶段。在所有 Dialplan 解析完成后，Channel 状态才进入执行（Standard EXECUTE，第 12 行）阶段。理解这一点是非常重要的，我们后面会再详细说明，但是在这里要记住路由查找（Routing 或 Parsing）和执行（Executing）分别属于一路通话的不同阶段。当 Channel 状态进入执行阶段后，它才开始依次执行所有的 Action。log 的作用就是将信息写到日志（Log）中，它的第一个参数是 loglevel，就是 Log 的级别，不同的级别在彩色的终端上能以不同的颜色显示，日志的级别有以下几种（其中数越大显示越详细）：

```
0 - CONSOLE
1 - ALERT
2 - CRIT
3 - ERR
4 - WARNING
5 - NOTICE
6 - INFO
7 - DEBUG
```

读者应该可以看到彩色的 Log 了，同时也会看到用 $ 表示的 Chan Var 被替换成了相应的值。细心的读者还会看到，在这次实验中我们特意增加了几个 Action。一个 Action 通常有两个属性（Attribute）：一个是 Application，代表要执行的 App；另一个是 data，就是 App 的参数，当 App 没有参数时，data 也可以省略。

一个 Action 必须是一个合法的 XML 标签，在前面读者看到的 context、extension 等都是成对出现的，如 <extension></extension>。但由于 Action 比较简单，一般采用简写的形式来关闭标签，即 <action/>。注意大于号前面的“/”，如果不小心漏掉，在 reloadxml 时将会

出现类似“+OK [[error near line 3371]: unexpected closing tag </condition>]”的错误，而实际的错误位置又通常不是出错的那一行。这是在编辑 XML 文件时经常会遇到的问题，因此较难于查找。因此在修改时要多加小心，并推荐使用具有语法高亮功能的编辑器[⊖]来编辑。

读到这里，读者或许还有疑问：既然我们在 info App 的输出里没看到 destination_number 这一变量，那为什么我们可以引用它呢？它到底是从哪里来的？是这样的：它在 info 中的输出是 Caller-Destination-Number，但你在引用的时候就需要使用 destination_number。还有一些变量，在 info 中的输出是 variable_xxxx，如 variable_domain_name，而实际引用时要去掉“variable_”前缀。不要紧张，这里有一份简明的对照表（见表 6-2）。完整的对照表请参见：http://wiki.freeswitch.org/wiki/Channel_Variables#Info_Application_Variable_Names_.28variable_xxxx.29。

表 6-2 部分 Info 中显示的变量与通道变量的对应关系

Info variable Name	Channel Var Name	说 明
Channel-State	state	当前 Channel 的状态
Channel-State-Number	state_number	当前 Channel 的状态整数值
Channel-Name	channel_name	Channel 名称
Unique-ID	uuid	Channel 的 UUID
Call-Direction	direction	呼叫方向，Inbound 或 Outbound
Answer-State	state	应答状态
Channel-Read-Codec-Name	read_codec	读 Codec
Channel-Read-Codec-Rate	read_rate	读采样率
Channel-Write-Codec-Name	write_codec	写 Codec
Channel-Write-Codec-Rate	write_rate	写采样率
Caller-Username	username	用户名
Caller-Dialplan	dialplan	使用的 Dialplan，如 XML、lua、enum、lcr 等
Caller-Caller-ID-Name	caller_id_name	主叫名称
Caller-Caller-ID-Number	caller_id_number	主叫号码
Caller-ANI	ani	主叫的 ANI，一般与 caller_id_number 同
Caller-ANI-II	aniii	主叫的 ANI II，如果有的话
Caller-Network-Addr	network_addr	主叫的 IP
Caller-Destination-Number	destination_number	被叫号码
Caller-Unique-ID	uuid	Channel UUID
Caller-Source	source	呼叫来源，如 mod_sofia、mod_freetdm 等
Caller-Context	context	Dialplan Context
Caller-RDNIS	rdnis	原被叫号码，一般用于呼叫转移时
Caller-Channel-Name	channel_name	Channel 的名字

⊖ 借助语法高亮通常能很快发现错误。这些编辑器包括 Windows 平台上的 Notepad++、UNIX 类平台上的 Vim 或 Emacs 等，笔者常用的是 Sublime Text 2，它是跨平台的。

（续）

Info variable Name	Channel Var Name	说 明
Caller-Profile-Index	profile_index	Profile Index
Caller-Channel-Created-Time	created_time	创建时间
Caller-Channel-Answered-Time	answered_time	应答时间
Caller-Channel-Hangup-Time	hangup_time	挂机时间
Caller-Channel-Transfer-Time	transfer_time	转移时间

6.1.5 测试条件

上面我们已经看到了最简单的测试条件，<condition field="destination_number" expression="^1234$">，它使用正则表达式匹配测试一个变量是否满足预设的正则表达式。大部分的测试都是针对被叫号码（destination_number）的，但你也可以对其他变量进行测试，如 IP 地址（注意由于在正则表达式中“.”具有特殊意义，所有需要用“\”进行转义）就可这样测试：

```
<condition field="network_addr" expression="^192\.168\.7\.7$">
```

表 6-3 列出了大部分的测试条件（Condition）。

表 6-3 Dialplan 中的测试条件

变 量	说 明
context	Dialplan 当前的 Context
rdnis	被转移的号码（在呼叫转移中设置）。
destination_number	被叫号码
dialplan	Dialplan 模块的名字，如 XML、YAML、inline、asterisk、enum 等
caller_id_name	主叫（来电显示）的名称
caller_id_number	主叫号码
ani	主叫的自动号码识别（Automatic Number Identification）
aniii	主叫类型，如投币电话（payphone）
uuid	本 Channel 的唯一标志
source	呼叫源，来自哪一个 FreeSWITCH 模块，如 mod_portaudio、mod_sofi 等
chan_name	Channel 名字，如 portaudio/1234
network_addr	主叫的 IP 地址
year	当前的年，0 ~ 9999
yday	一年中的第几天，1 ~ 366
mon	月，1 ~ 12
mday	日，1 ~ 31
week	一年中的第几周，1 ~ 53
mweek	本月中的第几周，1 ~ 6
wday	一周中的第几天，1 ~ 7（周日等于 = 1，周 1 等于 = 2，有点奇怪吧？）

（续）

变 量	说 明
hour	小时，0 ~ 23
minute	分，0 ~ 59
minute-of-day	一天中的第几分钟，(1 ~ 1440)（午夜 = 1, 1 点 = 60, 中午 = 720)

当然，除此之外，测试还接受用户在用户目录中设置的变量（参见第 5 章），但要注意必须使用 ${} 对变量进行引用。如下面的 toll_allow 就是在用户目录中设置的：

```
<condition field="${toll_allow}" expression="international">
```

如果在你的用户目录中设置了以下变量，则可以在 Dialplan 中通过下面的 condition 一行进行相关测试。

用户目录中的设置：

```
<user id="1000">
  <variables>
    <variable name="my_test_var" value="my_test_value"/>
  </variables>
</user>
```

Dialplan 中的测试条件：

```
<condition field="${my_test_var}" expression="^my_test_value$">
```

一般来说，测试条件不可以嵌套[⊖]，但可以迭加，如下面的例子并不能达到你预期的效果：

```
<extension name="Testing Stacked Conditions">
  <condition field="network_addr" expression="^192\.168\.7\.7$">
    <condition field="destination_number" expression="^1234$">
        <action application="log" data="INFO Hahaha, I know you called
${destination_number}"/>
    </condition>
  </condition>
</extension>
```

但以下配置是正确的：

```
<extension name="Testing Stacked Conditions">
  <condition field="network_addr" expression="^192\.168\.7\.7$"/>
  <condition field="destination_number" expression="^1234$">
    <action application="log" data="INFO Hahaha, I know you called ${destination_
number}"/>
  </condition>
</extension>
```

⊖ FreeSWITCH 于 2012 年底加入了嵌套 Dialplan 的支持（http://jira.freeswitch.org/browse/FS-4935）。这使得它的功能更加强大，但如其作者所说，这对于写文档的人或写书的人而言就不是个好消息了（bad news for documenters and authors）。限于本书的篇幅，在这里就不多介绍了。读者只要理解了本章的内容，如果需要这么复杂条件的读者请自行参阅 http://wiki.freeswitch.org/wiki/FreeSwitch_Dialplan_XML 上的 Nested_Conditions 部分。

注意上述代码中的第一个 condition，它的标签是在一行内用简写的形式关闭的，实际上它等价于：

```
<condition field="network_addr"
expression="^192\.168\.7\.7$"></condition>
```

所以，它与下面测试 destination_number 的 condition 是迭加的关系。因此就构成了一个简单的“逻辑与”的关系，即只有在 IP 地址和被叫号码两个条件都匹配的情况下才执行此处的 Action，否则就跳过本 extension，继续解析下一项。因此我们说这两个“平行”的 condition 是“逻辑与”的关系。

除此之外，两个迭加在一起的 condition 还可以构成其他关系，使你可以只用 XML 就能完成比较复杂的路由配置，而无须编程。break 参数就是用于实现这个功能的，它有以下几个值（为方便讨论，假设两个条件分别为 A 和 B）：

- on-false。在第一次匹配失败时停止（但继续处理其他的 extension），这是默认的配置。结果相当于 A and B。
- on-true。在第一次匹配成功时停止（但会先完成对应的 Action，然后继续处理其他的 extension），不成功则继续，所以结果相当于 ((not A) and B)
- always。不管是否匹配，都停止。
- never。不管是否匹配，都继续。

通过使用 break 参数，你可以写出类似 if-then-else 的结构。如上面的例子（因为没有 break 参数，所以默认是 on-false）就是（“//”是注释符）：

```
// 伪代码, =~ 表示正则表达式匹配
if(network_addr =~ /^192\.168\.7\.7$/) then
    if(destination_number =~ /^1234$/) then
        //wirte log
    end
end
```

我们再来看一个例子，假设你呼叫 1234，如果来自 192.168.7.7 这个 IP，就播放早上好，如果来自 192.168.7.8，就说晚上好，这时你可以这样设置：

```
<extension name="Testing Stacked Conditions">
  <condition field="destination_number" expression="^1234$">
  <condition field="network_addr" expression="^192\.168\.7\.7$"/>
    <action application="playback" data="good-morning.wav"/>
  </condition>
</extension>

<extension name="Testing Stacked Conditions">
  <condition field="destination_number" expression="^1234$">
  <condition field="network_addr" expression="^192\.168\.7\.8$"/>
    <action application="playback" data="good-night.wav"/>
  </condition>
</extension>
```

通过使用 break 参数，你就可以将两种情况写到一个 extension 里：

```
<extension name="Testing Stacked Conditions">
  <condition field="destination_number" expression="^1234$"/>
  <condition field="network_addr" expression="^192\.168\.7\.7$" break="on-true">
    <action application="playback" data="good-morning.wav"/>
  </condition>
  <condition field="network_addr" expression="^192\.168\.7\.8$">
    <action application="playback" data="good-night.wav"/>
  </condition>
</extension>
```

这种情况跟上面两个 extension 的情况是等价的。虽然看起来不如第一种情况容易理解，但它把相似的功能逻辑上放到一个 extension 里，却比较直观。如果你从 192.168.7.7 上呼叫 1234，它会首先匹配 1234，进而匹配网络地址，匹配成功，便不再往下进行。但是，它会先把已经匹配到的条件中的 Action 执行完毕，所以播放 good-morning.wav。若你从 192.168.7.8 上呼叫，则它不能匹配 192.168.7.7，但由于 break 参数的值是 on-true，所以在这里，它不会中止，而是继续尝试匹配下面的 condition，进而播放 good-night.wav。所以，它相当于：

```
if(destination_number =~ /^1234$/) then
    if(network_addr =~ /^192\.168\.7\.7$/) then
        // play good morning
    else if(network_addr =~ /^192.168.7.8$/) then
        // play good night
    end
end
```

always 和 never 不常用，发挥你的想象力，可以创造出类似 if a then b end 或 if c then d end 之类的条件。

6.1.6 动作与反动作

除使用 condition 的 break 机制来完成复杂的条件以外，你还可以使用“反动作”（Anti-Action）来达到类似的目的，如：

```
<extension name="Anction and Anti-Action">
  <condition field="destination_number" expression="^1234$"/>
  <condition field="network_addr" expression="^192\.168\.7\.7$">
    <action application="playback" data="good-morning.wav"/>
    <anti-action application="playback" data="good-night.wav"/>
  </condition>
</extension>
```

上述代码它说明，如果呼叫来自 192.168.7.7，则播放 good-morning.wav，否则播放 good-night.wav（不管是不是来自 192.169.7.8）。因此，读者可以看到，它没有 condition 条件那么强大，但在简单的条件下也经常使用，它相当于：

```
if(destination_number =~ /^1234$/) then
    if(network_addr =~ /^192\.168\.7\.7$/) then
        // play good morning
    else
        // play good night
    end
end
```

6.1.7　工作机制深入剖析

在进一步了解 Dialplan 的工作机制之前，我们先来看一下 Channel 的状态机，如图 6-1 所示。



图 6-1　Channel 状态机

当新建（NEW）一个 Channel 时，它首先会进行初始化（INIT），然后进入路由（ROUTING）阶段，也就是我们查找解析 Dialplan 的阶段。在这里，要注意一个专门的术语——Hunting（在传统的交换机里，它译为选线，在这里我就译为选路吧）。找到合适的路由入口后，Hunting 会执行（EXECUTE）一系列动作，最后无论哪一方挂机，都会进入挂机（HANGUP）阶段。后面的报告（REPORTING）阶段一般用于进行统计、计费等，最后将 Channel 销毁（DESTROY），释放系统资源。

在 EXECUTE 状态，可能会发生转移（Transfer，该转移跟我们通常说的呼叫转移不太一样），它可以转移到同一 context 下其他的 extension，或者转移到其他 context 下的 extension，但无论发生哪种转移，都会重新进行路由，也就是重新进入 ROUTING 阶段（图 6-1 中虚线部分），重新 Hunt Dialplan。

我们在前面的章节也讲过，一定要记住 ROUTING 和 EXECUTE 是属于两个不同阶段的，只有 ROUTING 完毕后才会进行 EXECUTE 阶段的操作。当一个 Channel 进入 ROUTING 阶段时，它首先会到达 Dialplan（英文叫 Hit the Dialplan），然后对预设的 Dialplan 进行解析（是的，每个电话都会重新解析 Dialplan），解析 Dialplan 的这一过程称为 Parsing 或 Hunting。解析完毕（成功）后，会得到一些 Action，然后 Channel 进入 EXECUTE 阶段，依次执行所有的 Action。

作为一个例子，我们用另一种方式实现类似上面的 good-morning/good-night 中的例子，但这一次稍有不同：用户呼叫 1234，如果来自 192.168.7.7，则播放 good-morning.wav；如果来自 192.168.7.8，则播放 good-night.wav；否则，什么都不做。（注意：**这个例子是错误的，它不会达到你预期的结果**，但这是新手常常犯的错误，而遇到这种情况，大多数新手都会以为 FreeSWITCH 出问题了）：

```
<extension name="Testing Hunting and Executing" continue="true">
  <condition>
    <action application="set" data="greeting=no-greeting.wav"/>
  </condition>
</extension>

<extension name="Testing Hunting and Executing" continue="true">
  <condition field="network_addr" expression="^192\.168\.7\.7$">
    <action application="set" data="greeting=good-morning.wav"/>
  </condition>
</extension>
```

```
<extension name="Testing Hunting and Executing" continue="true">
  <condition field="network_addr" expression="^192\.168\.7\.8$">
    <action application="set" data="greeting=good-night.wav"/>
  </condition>
</extension>

<extension name="Testing Hunting and Executing">
  <condition field="destination_number" expression="^1234$"/>
  <condition field="${greeting}" expression="^good">
    <action application="playback" data="${greeting}"/>
  </condition>
</extension>
```

首先，也许你已经看到，我们在前三个 extension 中都增加了一个参数 continue="true"。如果没有该参数，则默认为 false。在默认的情况下（试想一下前面的例子），在 Dialplan 的 Hunting 阶段，一旦根据前面介绍的 condition 匹配规则找到对应的 extension，就执行相应的 Action，而不会再继续查找其他的 extension 了，不管后面的 extension 是否有可能匹配。

但有些情况下，Dialplan 中会有多个 extension 满足匹配规则，而我们希望所有对应的 Action 都能得到执行，这时我们就要使用 continue="true" 参数了。

此外，我们这次还在第一个 extension 中用了一个空的 condition，这个空的 condition 没有匹配规则，因此它被认为匹配所有规则，故称其为绝对条件（Absolute Condition）。所以从表面上看，它等价于：

```
$greeting = no-greeting.wav

if (network_addr =~ /^192\.168\.7\.7$/) then
    $greeting = good-morning.wav
else if (network_addr =~ /192\.168\.7\.8$/) then
    $greeting = good-night.wav
end

if (destination_number =~ /^1234$/ and $greeting =~ /^good/) then
    play $greeting
end
```

但实际上两者不是等价的。原因在于，对 Dialplan 的 Hunting 和 Executing 分属于不同的阶段。在 Hunting 阶段，只解析 Dialplan，并不执行任何动作，而是将所有满足条件的 Action 都放到一个动作列表（队列）中，待呼叫流程进行到 Executing 阶段时，再依次执行动作列表中的动作。所以上述的 XML 实际上等价于：

```
$action_list[0] = "$greeting = no-greeting.wav"

if (network_addr =~ /^192\.168\.7\.7$/) then
    $action_list[1] = "$greeting = good-morning.wav"
else if (network_addr =~ /^192\.168\.7\.8$/) then
    $action_list[1] = "$greeting = good-night.wav"
end

if (destination_number =~ /^1234$/ and $greeting =~ /^good/) then
    $action_list[2] = "play $greeting"
```

```
end

// 开始执行命令列表中的命令

foreach $action in $action_list do
    execute $action
end
```

到了这里你应该找出问题所在了。因为你已经看到，在 Hunting 阶段，greeting 这个变量始终是空值，因为没有任何动作设置这个值。所以在测试 greeting 与正则表达式 /^good/ 是否匹配时，永远是不通过的。因此最后的动作列表中，也只有两个动作 action_list[0] 和 action_list[1]。虽然在执行阶段 $greeting 的值最终会被设置为 no-greeting、good-morning 或 good-night，但它再也不会回到 Hunting 阶段了（使用 transfer 除外），因而也不可能有机会通过 /^good/ 测试条件进而再执行真正的 playback 动作以播放声音。

关于 XML Dialplan 就介绍到这里。因为它本身就比较复杂，所以可能有些读者读到这里依然不是很明白，但是限于篇幅我们也只能介绍到这种程度。读者可反复阅读本节，理清前后关系，若依然不明白，笔者建议大家这样做：把这些 XML 放到你的机器上，观察 Log 的输出，并改一下某些参数对比一下 Log 有什么不同（如把 true 改成 false，把 IP 换成你自己的 IP 地址等）。

笔者在这里写了一个错误的例子，故有责任把它改正，请看下一节。

6.1.8　内联执行

在 6.1.7 节我们讲了一个错误的例子。原因是在 Dialplan 的 Hunting 阶段不执行 Action，但你却认为它应该执行。当然，如果你学会了看 Log，完全可以从 Log 中看出这一问题。但实际上，大多数的新手都不会仔细地去看 Log，而且即使仔细看了，由于经验比较少，也不一定能找到问题所在。FreeSWITCH 的开发者和老手们为了解决这个问题，便在 Action 上增加了一个 inline 属性。

好了，现在我们就把所有带 set 的 Action 都加上 inline="true" 这一属性，问题就迎刃而解了，如：

```
<action inline="true" application="set" data="greeting=no-greeting.wav"/>
```

是的，上述就是你期望的结果。在 Hunting 阶段，如果发现带有 inline 的 Action，FreeSWITCH 便会直接执行它，而不用等到 EXECUT 阶段。

改好后，认真对比一下 Log 输出与前面有什么不同。相信到这里你基本上完全理解 Dialplan 了。

当然，并不是所有的 App 都能用 inline 执行。以 inline 方式执行的 App 实际上相当于不遵守正常的执行流程，因而使用起来会有一些限制——适合 inline 执行的 App 必须能很快地执行，一般只是很快地存取某个变量，并且不能改变当前 Channel 的状态。满足这样条件的 App 有：check_acl、eval、event、export、enum、log、presence、set、set_global、lcr、set_profile_var、set_user、sleep、unset、nibblebill、verbose_events、cidlookup、curl、

easyroute、odbc_query。

当然，inline 参数也不是解决所有问题的万能钥匙，由于它会打乱执行顺序，所以使用不当也可能会产生非预期的结果。考虑下面的例子：

```
<action inline="true" application="set" data="var=1"/>
<action application="info"/>
<action inline="true" application="set" data="var=2"/>
<action application="info"/>
```

乍一看，第一个 info 输出结果中应该有 variable_var=1，第二个 info 输出结果中 variable_var=2。而实际上，在最后的输出中，两个 info 的输出结果都会显示 variable_var=2。想一想，为什么[⊖]？

6.1.9 实例解析

以上的论述应该涵盖了 Dialplan 的大部分概念，当然，要活学活用，还需要一些经验。下面，我们讲几个真实的例子，来帮助读者加深一下印象。这些例子大部分来自默认的配置文件。

1. Local_Extension

我们要看的第一个例子是 Local_Extension。FreeSWITCH 默认的配置提供了 1000 ~ 1019 共 20 个 SIP 账号，密码都是 1234。FreeSWITCH 通过以下 Dialplan 可以将来话路由到这些本地的号码。conf/dialplan/default.xml 中的 Local_Extension 部分如下：

```
<extension name="Local_Extension">
    <condition field="destination_number" expression="^(10[01][0-9])$">
    <!-- many actions，此处暂时省略 -->
    </condition>
</extension>
```

这个框架说明，用正则表达式 (10[01][0-9])$ 来匹配被叫号码，它匹配所有 1000 ~ 1019 这 20 个号码。

这里我们假设在 SIP 客户端上，用 1000 和 1001 分别注册到了 FreeSWITCH 上，则 1000 呼叫 1001 时，FreeSWITCH 会建立一个 Channel，该 Channel 构成一次呼叫的 a-leg（a 腿）。初始化完毕后，Channel 进入 ROUTING 状态，即到达 Dialplan。由于被叫号码 1001 与这里的正则表达式匹配，所以接下来会执行下面这些 Action。另外，由于我们在正则表达式中使用了“( )”，因此匹配结果会放入变量 $1 中，因此在这里 $1=1001。

下面我们再依次来看一下刚才我们省略的“ many actions”里面的内容，先从以下两行开始看：

```
<action application="set" data="dialed_extension=$1"/>
<action application="export" data="dialed_extension=$1"/>
```

⊖ 因为 set 是以 inline 方式执行的，而 info 要等到 EXECUTE 阶段才被执行，所以实际执行的顺序是 set → set → info → info，很显然，执行到 info 时变量的值已经是 2 了。

其中，set 和 export 都是设置一个变量，该变量的名字是 dialed_extension，值是 1001。

关于 set 和 export 的区别我们在第 5 章已经讲过了。这里再重复一次：set 是将变量设置到当前的 Channel 上，即 a-leg。而 export 则除具备 set 的功能外，也将变量设置到 b-leg 上。当然，这时 b-leg 还不存在。所以在这里它对该 Channel 的影响与 set 其实是一样的。因此，使用 set 完全是多余的。但是除此之外，export 还设置了一个特殊的变量，叫 export_vars，它的值是 dialed_extension。所以，实际上面的第二行就等价于下面的两行：

```
<action application="set" data="dialed_extension=$1"/>
<action application="set" data="export_vars=dialed_extension"/>
```

接着往下看：

```
<!-- bind_meta_app can have these args <key> [a|b|ab] [a|b|o|s] <app> -->
<action application="bind_meta_app" data="1 b s execute_extension::dx
XML features"/>
<action application="bind_meta_app" data="2 b s
record_session::$${recordings_dir}/${caller_id_number}.${strftime(%Y-%m-%d-%H-%M-%S)}.wav"/>
<action application="bind_meta_app" data="3 b s execute_extension::cf
XML features"/>
<action application="bind_meta_app" data="4 b s
execute_extension::att_xfer XML features"/>
```

其中，bind_meta_app 的作用是在该 Channel 上绑定 DTMF。上面四行分别绑定了 1、2、3、4 四个按键，它们都绑定到了 b-leg 上。注意，这时候 b-leg 还不存在。所以请记住这里，后面我们再讲。继续往下看：

```
<action application="set" data="ringback=${us-ring}"/>
```

此处，设置回铃音是美音（不同国家的回铃音是有区别的），${us-ring} 的值是在 vars.xml 中设置的。接下来：

```
<action application="set" data="transfer_ringback=$${hold_music}"/>
```

以上是设置如果发生呼叫转移时，用户听到的回铃音[⊖]。

下面这些变量会影响呼叫流程，关于它们的详细说明参见下文的 bridge 部分。

```
<action application="set" data="call_timeout=30"/>
```

call_timeout 是设置呼叫超时的变量。

```
<action application="set" data="hangup_after_bridge=true"/>
<!--<action application="set"
data="continue_on_fail=NORMAL_TEMPORARY_FAILURE,USER_BUSY,NO_ANSWER,TIMEOUT,
NO_ROUTE_DESTINATION"/> -->
<action application="set" data="continue_on_fail=true"/>
```

继续往下看：

```
<action application="hash"
    data="insert/${domain_name}-
```

⊖ 它跟 ringback 的区别是，ringback 是真正的回铃音，是在电话 answer 之前放的；而如果电话已经被接听（如到了 IVR），那么再发生转移时，就放 transfer_ringback 指定的声音。

```
    call_return/${dialed_extension}/${caller_id_number}"/>
    <action application="hash"
        data="insert/${domain_name}-
    last_dial_ext/${dialed_extension}/${uuid}"/>
        <action application="hash"
        data="insert/${domain_name}-last_
        dial_ext/${called_party_callgroup}/${uuid}"/>
    <action application="hash"
        data="insert/${domain_name}-last_dial_ext/global/${uuid}"/>
```

hash 是内存中的哈希表数据结构。它可以设置一个键 - 值对（Key-Value Pair）。如上面最后一行向 ${domain_name}-last_dial_ext 这个哈希表中插入一个 global 键，它的值是 ${uuid}，即本 Channel 的唯一标志[⊖]。

不管是上面的 set，还是 hash，都是保存一些数据为后面做准备的。不同的是 set 将变量存绑定到 Channel 上，以通道变量的形式存在，而 hash 保存到内存的哈希表数据结构中。

继续往下看，还是设置通道变量：

```
    <action application="set"
    data="called_party_callgroup=${user_data(${dialed_extension}@${domain_name}
var callgroup)}"/>
    <!--<action application="export"
    data="nolocal:sip_secure_media=${user_data(${dialed_extension}@${domain_name}
var sip_secure_media)}"/>-->
```

上面最后一行默认是注释掉的，因此不起作用。nolocal 的作用我们之前也讲过，它告诉 export 只将该变量设置到 b-leg 上，而不要设置到 a-leg 上。

下面，还是往哈希表中插入数据：

```
    <action application="hash"
        data="insert/${domain_name}-
    last_dial/${called_party_callgroup}/${uuid}"/>
```

再往下看，终于到了一个干实事的地方了：

```
    <action application="bridge"
        data="{sip_invite_domain=$${domain}}user/${dialed_extension}@${domain_name}"/>
```

这里，bridge 是最关键的部分。其实上面除 bridge 以外的 Action 都可以省略，只是会少一些功能而已（如同组代答、监听等）。

bridge 相当于一座桥，它的作用就是把两条腿 1000 和 1001 给桥接起来。在这里，为了能连接到 1001，FreeSWITCH 作为一个 SIP UAC[⊜]，向 1001 这个 SIP UA（UAS）发起一个 INVITE 请求，并建立一个新 Channel，就是我们的 b-leg。1001 开始振铃，bridge 把回铃音传回到 1000，因此 1000 就能听到回铃音（如果 1001 有自己的回铃音，则 1000 也能听到，

⊖ 每一个 Channel 都有一个唯一标志，称为 Channel UUID。该变量在全局代答（又叫代接，见 2.2 节）业务中会用到，使用时，其他分机可以通过拨打一个号码查找到这个键值，进而可以“代接”。

⊜ 我们会在第 7 章讲到这些 UA 的概念。用户 1000 其实是一个 SIP UA（UAC），它向 FreeSWITCH（作为 UAS）发送一个 INVITE 请求。然后 FreeSWITCH 建立一个 Channel，从 INVITE 请求中找到被叫号码（destination_number=1001），然后在 Dialplan 中查找 1001。

否则将会听到默认的回铃音 ${us-ring})。

当然，实际的情况比我们所说的要复杂，因为在呼叫之前，FreeSWITCH 首先要查找 1001 这个用户是否已经注册，否则，会直接返回 USER_NOT_REGISTERED，而不会建立 b-leg。

bridge 的参数是一个标准的呼叫字符串（Dial string），以前我们也讲过。domain 和 domain_name 都是预设的变量，默认就是服务器的 IP 地址。user 是一个特殊的 Endpoint，它指本地用户。所以这里的呼叫字符串翻译出来就是（这里假设 IP 是 192.168.7.2）：

```
{sip_invite_domain=192.168.7.2}user/1001@192.168.7.2
```

其中，"{ }"里是设置通道变量。由于 bridge 在这里要建立 b-leg，因此这些变量只会建立在 b-leg 上。这与 set 是不一样的，但它等价于下面的 export：

```
<action application="export" value="nolocal:sip_invite_domain=192.168.7.2"/>
<action application="bridge" value="user/1001@192.168.7.2"/>
```

好了，到此为止电话路由基本上就完成了，我们已经建立了 1000 到 1001 之间的呼叫，1001 已经开始振铃，就等着有人来接电话了。接下来可能会有以下几种情况：

- 被叫应答
- 被叫忙
- 被叫无应答
- 被叫拒绝
- 其他情况……

我们先来看一下被叫应答的情况。1001 接电话，与 1000 畅聊。在这个时候 bridge 一直是阻塞的，也就是说，bridge 这个 App 会一直等待 b-leg（1001）挂机（或者其他错误）后才返回，这时才有可能继续执行下面的 Action。好吧，让我们先休息一下，等他们两个聊完吧。

最后，无论哪一方挂机，bridge 就算结束了。如果 1000（主叫）先挂机，则 FreeSWITCH 会将挂机原因（Hangup Cause，一般是 NORMAL_RELEASE，即正常释放）发送给 1001，同时释放 b-leg。由于 a-leg 已经没了，Dialplan 就再也没有往下执行的必要，因此会产生计费信息，并销毁 a-leg。

如果 1001 先挂机，b-leg 就这样消失了。但 a-leg 依然存在，所以还有戏看。

b-leg 会将挂机原因传到 a-leg。在 a-leg 决定是否继续往下执行之前，会检查一些变量，以决定该怎么做。其中，我们在前面设置了 hangup_after_bridge=true。它的意思是，如果 bridge 正常完成后，就挂机。因此，a-leg 到这里就释放了，它的挂机原因是参考 b-leg 得出的。

但由于种种原因 1001 可能没接电话，如 1001 可能会拒接（返回 CAlL_REJECTED，但某些 SIP UA 会在用户拒接时返回 USER_BUSY，隐藏真实原因）、忙（USER_BUSY）、无应答（NO_ANSWER 或 NO_USER_RESPONSE）等。出现这些情况时，FreeSWITCH 认为这是不成功的 bridge，因此就不管用 hangup_after_bridge 变量了（由于没有成功的 bridge，也就也没有"after"一说了）。这时候它会检查另一个变量 continue_on_fail。由于我们上面设置的 continue_on_fail=true，因此在 bridge 失败后会继续（continue）执行下面的 Action。

这里值得说明的是，通过给 continue_on_fail 不同的值，可以决定在什么情况下继续，如下面的设置将只在用户忙和无应答的情况下才会继续呼叫流程：

```
<action application="set"
data="continue_on_fail=USER_BUSY,NO_ANSWER"/>
```

其他可能的值有：NORMAL_TEMPORARY_FAILURE（临时故障）、TIMEOUT（超时，一般是 SIP 超时）、NO_ROUTE_DESTINATION（呼叫不可达）等。

当然，此处的 continue_on_fail=true 表示无论什么原因导致 bridge 失败（我们没法联系上 1001），我们都决定继续执行。言归正传，我们接着往下看：

```
<action application="answer"/>
```

Dialplan 执行到这里，answer App 首先会使 FreeSWITCH 给主叫 1000 回送应答消息，以建立真正的媒体流，准备对其播放声音。接下来：

```
<action application="sleep" data="1000"/>
<action application="voicemail" data="default ${domain_name}
${dialed_extension}"/>
```

sleep 表示暂停，1000 表示暂停的毫秒数（即 1 秒），然后转到 1001 的语音信箱[⊖]。另外，值得注意的是，FreeSWITCH 默认配置文件中是通过 transfer 加 loopback Endpoint 的方式转到 voicemail 的，我们还没有学到 loopback，因为这里我们学到的新东西已经够多了，所以为了方便说明，笔者直接改成了 voicemail App 的形式，它们在这里的作用是等价的。

2. 回声和延迟回声

关于回声，没什么需要多解释的，我们已经非常熟悉这一段了。如果拨 9196，就能听到自己的回声。Dialplan 如下：

```
<extension name="echo">
    <condition field="destination_number" expression="^9196$">
        <action application="answer"/>
        <action application="echo"/>
    </condition>
</extension>
```

与 echo 类似，delay_echo 可以使用回声有一定延迟，Dialplan 如下（其中 5000 是毫秒数）：

```
<extension name="delay_echo">
    <condition field="destination_number" expression="^9195$">
        <action application="answer"/>
        <action application="delay_echo" data="5000"/>
    </condition>
</extension>
```

3. 会议

下面的 Dialplan 可以将来话转入一个会议。其中，nb_conference 的意思是 Narrow Band

⊖ 语音信箱即 Voicemail，在国内用得不多，因此在本书中我们不讨论这部分内容。

Conference，就是默认的8000Hz的窄带电话会议。

```
<extension name="nb_conferences">
    <condition field="destination_number" expression="^(30\d{2})$">
        <action application="answer" data=""/>
        <action application="conference" data="$1-
                                              ${domain_name}@default"/>
    </condition>
</extension>
```

上面的正则表达式“^(30)$”匹配以30开头的4位数字的被叫号码。一般测试时都可以直接呼叫3000或3001。所有呼叫30xx（30+两位数字）的电话均可以进入一个会议（conference），大家可以畅所欲言。其中，conference是一个App，它的参数扩展开就是（假设呼叫3000，IP地址是192.168.7.2）：

```
3000-192.168.7.2@default
```

其中，3000-IP地址是会议的名字，@后面的default表示一个会议的profile，它定义了这个会议的相关参数。我们在后面还会专门讲到多人电话会议，在这里，读者只需要知道呼叫3000能进入一个会议就可以了。

4. 将通话的双方转入会议

我们再回头看看6.1.9节中的bind_meta_app 3这一行：

```
<action application="bind_meta_app" data="3 b s execute_extension::cf XML features"/>
```

bind_meta_app是一个App，它有一系列参数，其中，3表示绑定DTMF按键3；b表示要绑到b-leg上；s表示Same。即如果在b-leg上按下“*3”（必须先按“*”），则执行execute_extension这个App。execute_extension是临时去别的地方执行一些Dialplan指定的App，在这里它的参数是cf XML features，表示要去XML Dialplan的features这个Context中找一个匹配cf的extension。

假设1000呼叫1001，则1001摘机后可以与1000通话。这时，由于1001是b-leg，因而它可以通过按“*3”这个DTMF按键触发execute_extension动作，我们接下来看会发生什么。

features这个Context在conf/dialplan/features.xml中定义，我们可以很容易找到匹配cf的这个extension：

```
<!-- Used to transfer both legs into a conference -->
<extension name="cf">
    <condition field="destination_number" expression="^cf$">
        <action application="answer"/>
        <action application="transfer" data="-both
30${dialed_extension:2} XML default"/>
    </condition>
</extension>
```

首先，answer这个App在这里没什么用，因为呼叫已经应答了。transfer是一个App，它会将当前通话重新转移到ROUTING阶段，重新去Dialplan中进行路由。这里它的第一个参数是-both，表示要将它自己和a-leg都转到Dialplan中重新路由。路由的Dialplan类型

仍然是 XML，Context 是 Default，被叫号码 30${dialed_extension:2} 中的 dialed_extension 变量来自于 6.1.9 节中的：

```
<action application="export" data="dialed_extension=$1"/>
```

注意，上面是用 export 把变量设置到 b-leg 上的，实际上就是最初的被叫号码 1001，${dialed_extension:2} 的意思是从 1001 的第 2 个位置开始截取，一直截取到字符串结尾的字符串，由于位置是从 0 开始计数的，因而 1001:2 最终的结果就是 01，因而最终 transfer 的一行就等价于：

```
<action application="transfer" data="-both 3001 XML default"/>
```

其中，-both 表示将两条腿都分别转到 3001 这个 extension 上。所以，重新路由后就到了会议的情况。最终 1000 和 1001 都进入名为 3001-192.168.7.2 的会议。

6.2 inline Dialplan

inline Dialplan 称为内联拨号计划。首先，它与 6.1.8 节讲的 XML Dialplan Action 中的 inline 属性（内联执行）是完全不同的概念，如果读者觉得容易混，那么在这里可以暂时先把前面讲的 inline 的概念忘掉，等熟悉了本节的知识以后，再翻到 6.1.8 节进行比较。

XML Dialplan 支持非常丰富的功能，但在测试或编写程序时，我们经常用到一些临时的或者很简单的 Dialplan，如果每次都需要修改 XML，不仅麻烦，而且执行效率也会有所降低。所以，我们需要一种短小、轻便的 Dialplan 以便更高效地完成任务，inline Diaplan 便是因此而生的。而且，通过使用 inline Dialplan，可以很方便地在脚本中生成动态的 Dialplan 而无须使用复杂的 reloadxml 以及 mod_xml_curl 技术等。

与 XML Dialplan 不同，inline Dialplan 没有 Extension，也没有复杂的 Condition，只是像 XML Dialplan 中那样简单地叠加 Action。它有一种很紧凑的语法格式：

```
app1:arg1,app2:arg2,app3:arg3
```

从语法可以看出，它只是多个 App 以及其参数组成的字符串，App 之间用逗号分隔，而 App 与参数之间用冒号分隔。如果参数中有空格，则整个字符串都需要使用单引号引起来。在我们上面的例子中，通过拨打 9196 来找到对应的 XML dialplan，在这里我们可以直接在命令行上写出对应的 inline Dialplan 的形式：

```
originate user/1000 echo inline
originate user/1000 answer,echo inline
```

读到这里，你可能要问，它与 originate user/1000 &echo 有什么区别呢？在回答这个问题之前，我们需要先看一下 originate 的语法：

```
originate <call_url> <exten>|&<application_name>(<app_args>)
[<dialplan>] [<context>] [<cid_name>] [<cid_num>] [<timeout_sec>]
```

首先，它的第一个参数 call_url 是呼叫字符串。第二个参数可以是一个 exten（这个一会

再讲)，或者 & 加上一个 App，App 的参数要放到括号里，如：

```
originate user/1000 &echo
originate user/1000 &playback(/tmp/sound.wav)
originate user/1000 &record(/tmp/recording.wav)
```

这是最简单的形式，首先，originate 会产生一个 Channel，它会呼叫 user/1000 这个用户。请注意，这是一个单腿的通话，因此只有一个 Channel。但一个 Channel 有两端，一端是 1000 这个用户，另一端是 FreeSWITCH（我们上面已经说过，实现上是 1000 在跟 FreeSWITCH 的某个 App 通话，如这里的 echo、playback 或 record）。

在 user/1000 接电话后（严格说是收到它的 earlymedia 后），FreeSWITCH 即开始在该 Channel 上执行 & 后面的 App。但这种形式只能执行一个 App，如果要执行多个，就需要将电话转入 Dialplan，可以使用下面的形式：

```
originate user/1000 9196
originate user/1000 9196 XML default
```

上面两个命令的效果是一样的。它们的作用是，在 user/1000 接听电话后，电话的另一端（也就是 FreeSWITCH）需要对电话进行路由，在这里它要将电话路由到 9196 这个 Extension 上，第一条命令由于没有指定是哪个 Dialplan，因此它会在默认的 XML Dialplan 中查找，同时 XML Dialplan 需要一个 Context，它默认就是 default。它的效果跟直接用软电话拨打 9196 这个分机一样（不同的是呼叫方向的问题，这种情况相当于回拨）。

当然，除此之外，它还可以加一些可选的参数，用于指定来电显示（Caller ID）的名字（cid_name）和号码（cid_number），以及呼叫超时的秒数，如：

```
originate user/1000 9196 XML default 'Seven Du' 9196 30
```

当然，我们这里学了 inline Dialplan，所以你也可以用下面这种形式：

```
originate user/1000 echo inline
```

请注意，在 XML Dialplan 中，9196 是一个分机号，而 inline Dialplan 中的 echo 是一个 App，当然你也可以顺序执行多个 App。如下列第一条 originate 命令分别执行 answer（应答）、playback（播放语音）、record（录音），第二条则分别执行 playback 和 bridge：

```
originate user/1000 answer,playback:/tmp/pleace_leave_a_message.wav,record:/
tmp/recording.wav inline
originate user/1000 playback:/tmp/beep.wav,bridge:user/1001 inline
```

有时候，App 的参数中可能会有逗号，因而会与默认的 App 间的逗号分隔符相冲突，以下的 m 语法形式将默认的逗号改为 ^ 分隔（相当于转义：

```
originate user/1000
'm:^:playback/tmp/beep.wav^bridge:{ignore_early_media=true,originate_caller_
id_number=1000}user/1001'
inline
```

其中，“m:^”表示将 App 间的分隔符临时改为了“^”，所以后面的 bridge 中的参数中的逗号就不会误认为是 App 间的分隔符了。

当然你也可以把 inline Dialplan 用在任何需要 Dialplan 的地方，如（以下两行实为一行）：

```
uuid_transfer 2bde6598-0f1a-48fe-80bc-a457a31b0055
'set:test_var=test_value,info,palyback:/tmp/beep.wav,record:/tmp/recording.wav'
```

上述命令会将一个 Channel 转入一个 inline Dialplan，它首先执行 set 以设置一个 test_var 的通道变量，然后执行 info 在日志中打印当前 Channel 的详细情况，接下来播放一个声音文件（如 beep 应该是“嘀”的一声），然后开始录音。

6.3 其他 Dialplan

除了以上介绍的 Dialplan 外，还有其他的 Dialplan 形式，要查看你的系统支持多少 Dialplan，可以使用如下命令：

```
freeswitch> show dialplan

type,name,ikey
dialplan,LUA,mod_lua
dialplan,XML,mod_dialplan_xml
dialplan,asterisk,mod_dialplan_asterisk
dialplan,enum,mod_enum
dialplan,inline,mod_dptools
```

上述命令列出了系统默认支持的 Dialplan 以及实现它们的模块。我们将在 16.2.7 节再讨论 LUA Dialplan，其他的 Dialplan 本书就不多介绍了。

6.4 常用的 Dialplan App

通过前面几节我们熟悉了各种 Dialplan，并知道 Dialplan 最终会找到并返回一组 App，实际的呼叫行为全部是由这些 App 来控制的。FreeSWITCH 中有超过 140 个 App，不过最常用的不是很多，在此我们对几个常用的、有代表性 App 进行讲解。

（1）set

set 用于设置一个通道变量，如：

```
<action application="set" data="my_var=123456"/>
```

（2）echo

echo 即回声，这个我们已经很熟悉了，它在调试的时候比较有用，例如：

```
<action applicatioin="echo"/>
```

（3）info

info 在调试的时候也比较有用，它会在日志中打印全部的通道变量，例如：

```
<action applicaiton="info"/>
```

（4）answer

answer 用于应答一路呼叫。FreeSWITCH 做被叫时，如果想给主叫方放音，则必须应答

后才可以（在 SIP 中是 200 消息，第 7 章会讲到），例如：

```
<action application="answer"/>
```

有些 App（如 conference 或 fifo 等）会隐含应答，因而不需要明确的应答。但有些 App（如 playback 和 ivr 等）则不会隐含应答，因而需要明确的应答（answer）。

在前面的章节我们也了解到，bridge 是一个比较特殊的 App，它会再发起一路呼叫 b-leg，如果 b-leg 应答，它会将 b-leg 的应答传递给 a-leg，进而给主叫用户发送应答消息。

一般来说，运营商都把应答时间作为计费的开始时间。所以在上面的例子中，如果 A 呼叫 B，只有 B 应答后，应答消息才会发给 A，因此只有 B 应答了才开始计费。

（5）bridge

bridge 负责桥接另一条腿（b-leg），它的参数是一个呼叫字符串：

```
<action application="bridge" data="user/1000">
```

bridge 操作是阻塞的，它会一直等到 b-leg 释放后才继续往下走。

（6）playback

playback 用于给 Channel 放音。比如，如果需要对主叫放音，我们可以这样使用 playback：

```
<action application="playback" data="/tmp/test.wav"/>
```

它的参数是声音文件的路径。一般的声音文件是 .wav 格式的。如果要播放多个文件，可以串联操作，如：

```
<action application="playback" data="/tmp/test1.wav"/>
<action application="playback" data="/tmp/test2.wav"/>
<action application="playback" data="/tmp/test3.wav"/>
```

也可以通过 file_string[⊖]协议来串联，如：

```
<action application="playback"
data="file_string:///tmp/test1.wav!/tmp/test2.wav!/tmp/test3.wav"/>
```

其中，“!”是文件名的分隔符，如果想使用其他分隔符，可以在 playback 之前使用以下 Chan Var 指定。另外，也可以指定多个文件间播放的时间间隔，如下代码会将文件名分隔符改成“|”，并把时间间隔改为 500 毫秒：

```
<action application="set" data="playback_delimiter=!"/>
<action application="set" data="playback_sleep_val=500"/>
```

另外，通过 mod_shout 模块，也可以支持本地或远程 HTTP 或 Shout Cast 服务器上的 .mp3 格式的文件，如：

```
<action application="playback" data="/tmp/test.mp3"/>
<action application="playback" data="http://localhost/test.mp3"/>
<action application="playback" data="shout://localhost/test.mp3"/>
```

在播放过程中，用户可以按“*”号键停止播放，或在播放前通过 Chan Var 选择其他按键完成此功能：

⊖ 参见 https://wiki.freeswitch.org/wiki/Mod_file_string

```
<action application="set" data="playback_terminators=1"/>
```

也可以使用如下方法取消按键中断的方式：

```
<action application="set" data="playback_terminators=none"/>
```

（7）sleep

sleep 用于设置可以等待 / 暂停的一段时间，单位默为毫秒：

```
<action application="sleep" data="1000"/>
```

（8）ring_ready

ring_ready 用于在 SIP 中给对方回 180 消息，即通知对方可以振铃了：

```
<action application="ring_ready" data="1000"/>
```

（9）pre_answer

pre_answer 用于在 SIP 中给对方回 183 消息，后续的 playback 之类的动作将作为早期媒体（Early Media）给对方发过去，如彩铃音：

```
<action application="ring_ready" data="1000"/>
<action application="playback" data="/tmp/music.wav"/>
```

注意，虽然 FreeSWITCH 可以把媒体发给对方，但如果在一定时间内（通常 60 秒）没有应答，对端通常也会挂断该通话。

（10）read

read 用于实现播放声音并且等待接收 DTMF 按键，它的格式是：

```
<min> <max> <sound file> <variable name> <timeout> <terminators>
```

其中：

- min：最少收号位数。
- max：最大收号位数。
- sound file：要播放的声音文件。
- variable name：收到用户按键后存到哪个变量里。
- timeout：等待每一位的输入超时毫秒数。
- terminators：收号小于 min 位时，按该键可以提前结束，通常是“#”。

例如，我们可以播放“您好，请输入身份证号，以 # 号结束（身份证号是 15 或 18 位的）”：

```
<action application="answer"/>
<action application="read" data="15 18 /tmp/input-id-card.wav
id_card_num 5000 #"/>
<action application="log" data="INFO ID Card Number: ${id_card_num}"/>
<action application="info"/>
<action application="hangup"/>
```

（11）play_and_get_digits

play_and_get_digits 与 read 类似，但它比 read 更高级，我们先来看一下它的参数（其

中，方括号内是可选参数）：

```
<min> <max> <tries> <timeout> <terminators> <file> <invalid_file>
<var_name> <regex>
[<digit_timeout>] ['<failure_ext> [failure_dp[failure_context]]']
```

它的作用也是播放一段语音等待收号，与 read 不同的是，它收到后会跟正则表达式比较，如果不匹配，则播放 invalid_file 所指定的声音（如“输入错误，请重新输入”），然后重新等待输入。重试次数由 tries 参数指定。

play_and_get_digits 大部分参数都与 read 的类似，具体如下：

- min：最少收号位数。
- max：最大收号位数。
- tries：重试次数。
- timeout：收齐全部位数的超时。
- terminators：结束符。
- file：要播放的声音文件。
- invalid_file：输入错误时播放的提示音文件。
- var_name：变量名。
- regex：正则表达式。
- digit_timeout（可选）：位间间隔。
- failure_ext（可选）：若最后输入错误则转到该 Extension。
- failure_dp（可选）：输入错误时转到的 Dialplan。
- failure_context（可选）：输入错误时转到的 Dialplan Context。

如上面输入身份证明的例子，我们可以允许用户重试 3 次，使用 play_and_get_digits 实现的方法如下：

```
<action application="play_and_get_digits"
    data="15 18 3 10000 # /tmp/input-id-card.wav /tmp/invalid_num.wav
    id_card_num (^\d{15}$)|(^\d{17}([0-9]|X)$)"/>
<action application="log" data="INFO ID Card Number: ${my_digits}"/>
```

6.5 在 Dialplan 中使用 API 命令

Dialplan 中一般执行的是 App，而且我们在第 4 章中介绍 App 与 API 的区别时也讲到过，API 本身与 Channel 或 Session 是不相关的，其是一个“第三者”。不过，在某些特殊情况下，在 Dialplan 中也需要调用某些 API 提供的能力，这可以通过类似变量引用的方法来实现，如：${status()}，它与变量引用的区别就是多了一对“()”，看起来类似于函数调用，括号内可以填写 API 命令的参数。

这些 API 调用一般是通过 set 来执行的，如：

```
<action application="set" data="api_result=${status()}"/>
<action application="set" data="api_result=${version()}"/>
```

```
<action application="set" data="api_result=${strftime()}"/>
<action application="set" data="api_result=${expr(1+1)}"/>
```

以上命令都很直观，它们都是执行一个 API，并将 API 输出的结果作为变量的值。其中，最后一行中的 expr 类似于 UNIX 中的 expr 命令，它计算一个表达式并输出结果，如：

```
freeswitch> expr 1+1

2
```

6.6 深入理解通道变量及相关操作

在 Dialplan 中，可以对通道变量（Channel Variable）进行各种操作。虽然有一些我们已经在前面的内容中提到过，但这里我们再来系统地学习一下。

比如，我们已经熟悉的 set App 可以给变量赋值：

```
<action application="set" data="my_var=my_value"/>
```

另外，export 可以对 a-leg 和 b-leg 同时赋值（即使此时 b-leg 还不存在）：

```
<action application="export" data="my_var=my_value"/>
```

以上两条命令都可以将 my_var 变量的值设置为 my_value。不同的是 set 程序仅会作用于当前的 Channel（a-leg），而 export 程序则会将变量设置到两个 Channel（a-leg 和 b-leg）上，如果当时 b-leg 还没有创建，则会在创建时设置。另外，export 也可以通过 nolocal 参数将变量值限制仅设置到 b-leg 上：

```
<action appliction="export" data="nolocal:my_var=my_value"/>
```

在实际应用中，如果 a-leg 上已经有一些变量的值（如 var1、var2、var3），但想同时把这些变量都复制到 b-leg 上，可以使用以下几种办法：

```
<action application="export" data="var1=$var1"/>
<action application="export" data="var2=$var2"/>
<action application="export" data="var3=$var3"/>
```

或者使用如下等价的方式：

```
<action application="set" data="export_vars=var1,var2,var3">
```

所以，set 也具有能往 b-leg 上赋值的能力。其实，它和 export 一样，都是操作 export_vars 这个特殊的变量。只不过 export 的语法更直观一些。

另外，我们也可以随时取消某些 Variable 的定义。取消 Variable 定义只需对它赋一个特殊的值——“_undef_”或使用 unset App，如，下面两行代码是等价的：

```
<action application="set" data="var1=_undef_"/>
<action application="unset" data="var1"/>
```

在实际使用时，还常常会用到截取 Variable 值的部分操作。在 FreeSWITCH 中，可以使用特殊的语法取一个通道变量值的子字符串，使用格式是“${var: 位置 : 长度 }”。其中

“位置”从 0 开始计数，若为负数则从字符串尾部开始计数；如果“长度”为 0 或小于 0，则会从当前“位置”一直取到字符串结尾（或开头，若“位置”为负的话）。例如 var 的值为 1234567890，那么各种取值方法见下面的例子：

```
${var}        = 1234567890        # 原始变量的值
${var:0:1}    = 1                 # 从第 0 个位置起，取 1 个字符
${var:1}      = 234567890         # 从第 1 个位置起，一直取到字符串结束
${var:-4}     = 7890              # 从倒数第 4 个位置起，一直取到字符串结束
${var:-4:2}   = 78                # 从倒数第 4 个位置起，取 2 个字符
${var:4:2}    = 56                # 从第 4 个位置起，取 2 个字符
```

6.7 小结

本章讲了 FreeSWITCH 中的几种拨号计划并重点讲解了 XML 拨号计划。XML 的描述能力很强，因而便于写出比较复杂的拨号计划。事实上，FreeSWITCH 默认的配置在这里已经发挥得淋漓尽致了。读者如果有时间可以好好地看一看。另外，在以后的章节中我们也会用到默认拨号计划中的例子。

拨号计划有三个核心要素：Dialplan、Context 和 Extension。系统中有一些 Endpoint 模块，这些模块一般都能配置相关的拨号计划以便对来话进行处理和路由。某些 App 或 API，如 transfer 和 uuid_transfer 等，需要的参数也是 Dialplan。在不同的场景下这些参数也会有默认值，如：

```
<action application="transfer" data="1234"/>
```

则表示 transfer 到 1234 这个 Extension，使用 XML Dialplan，Context 为 default，实际上它等价于：

```
<action application="transfer" data="1234 XML default"/>
```

理解了这些要素和参数，就可以举一反三，写出更强大的呼叫流程了。

通过对本章的学习中大家可以看到，不管是 XML Dialplan 还是 inline Dialplan，拨号计划的最终目的就是为了给我们返回一组 App 以及它们的参数。一旦返回，拨号计划就完成了它的使命，实际呼叫的应答、放音、收号，录音等行为全部是由这些 App 后续控制的。当然，这些 App 也可以在适当的时候进行转移（transfer），从而要求重新进行路由（见图 6-1 中的 Transfer 部分），以获取一组新的 App，进而执行新的动作。

execute_extension 与 transfer 类似，都需要 Dialplan 的“三要素”作为参数，但不同的是，前者是临时执行一些 Dialplan 指定的 App，它不会重新进入 ROUTING 阶段。

另外，我们也讲了 Dialplan 中一些常用的 App 以及在 Dialplan 中执行 API 的方法和技巧，灵活掌握这些可以使你在以后的配置维护和应用开发中事半功倍。

最后，我们深入讲解了通道变量的相关知识以及在 Dialplan 中对 Channel Variable 的一些常用操作方法，学完本章后读者可以自行尝试一下。

第 7 章 SIP 协议

经过对前面章节的学习，相信读者已经对 FreeSWITCH 的架构和基本概念了解得差不多了。在继续深入学习之前，我们需要了解 SIP。

SIP 协议是 FreeSWITCH 的核心协议，理解 SIP 协议对理解 FreeSWITCH 的设计理念和使用方法很有帮助。我们在第 1 章已经对 SIP 协议做了简单介绍，并与 H.323 协议做了简单对比。在本章，笔者会带领大家更深入地了解 SIP 的基本概念以及协议内容。本章不要求读者有 SIP 基础，但如果读者有一些 HTTP 协议的基础知识，对理解本章的内容还是很有帮助的。

7.1 SIP 协议基础

会话初始协议（Session Initiation Protocol）是一个控制发起、修改和终结交互式多媒体会话的信令协议。它是由 IETF（Internet Engineering Task Force，Internet 工程任务组）在 RFC 2543 [一]中定义的，最早发布于 1999 年 3 月，后来在 2002 年 6 月又发布了一个新的标准 RFC 3261 [二]。除此之外，还有很多相关的或是在 SIP 基础上扩展出来的 RFC，如关于 SDP 的 RFC 4566 [三]、关于会议的 RFC 4579 [四]等。

首先，我们先来看一下 SIP 协议中的基础知识和基本概念，为了便于理解，我们对照大家可能更熟悉的 HTTP 协议来进行讲解[五]。

[一] 参见 http://tools.ietf.org/html/rfc2543。

[二] 参见 http://tools.ietf.org/html/rfc3261。

[三] 参见 http://tools.ietf.org/html/rfc4566。

[四] 参见 http://tools.ietf.org/html/rfc4579。

[五] 当然，有的读者可能对 SIP 更熟悉一些。不管怎么样，HTTP 协议是当今互联网的基础，我们大家每天上网都离不开它。实际上，SIP 协议中有些功能就直接套用了 HTTP 中的定义，如 Digest 验证。而且，FreeSWITCH 支持使用 HTTP（Web）服务器提供动态的 XML 配置文件，因此 HTTP 协议也是有必要了解的。对比 HTTP 和 SIP 协议进行学习，通过比较它们的异同，可以比较容易理解其原理和机制，学起来事半功倍。另外，在第 11 章我们要讲到的 Event Socket（ESL）协议也是类似于 HTTP 与 SIP 的。

7.1.1 HTTP 与 SIP 协议基础

SIP 是一个基于文本的协议，这一点与 HTTP 和 SMTP 类似。我们来对比一组简单的 HTTP 请求与 SIP 请求。

```
HTTP:
GET /index.html HTTP/1.1
SIP:
INVITE sip:seven@freeswitch.org.cn SIP/2.0
```

两者类似，请求均有三部分组成：在 HTTP 请求中，GET 指明一个获取资源（文件）的动作，/index.html 则是资源的地址，最后 HTTP/1.1 是协议版本号；而在 SIP 中，INVITE 表示发起一次呼叫请求，seven@freeswitch.org.cn 为请求的地址，也称为 SIP URI 或 AOR（Adress of Record，用户的公开地址），第三部分的 SIP/2.0 也是版本号。其中，SIP URI 类似一个电子邮件地址，其格式为“协议 : 名称 @ 主机”。这里 SIP URI 格式中的“协议”与 HTTP 和 HTTPS 相对应，也有 SIP 和 SIPS 两种（后者是加密的，如 sips:seven@freeswitch.org.cn）；“名称”可以是一串数字的电话号码，也可以是字母表示的名称；而“主机”可以是一个域名，也可以是一个 IP 地址。

对于 HTTP 来说，好多人可能既熟悉又陌生。熟悉的是，天天听到看到它；陌生的是，好多人又不是真正了解 HTTP 协议。大家通常在使用浏览器上网时，浏览器与 Web 服务器间的通信使用的就是 HTTP 协议。只是，具体的协议文本和交互流程普通人看不见而已。在我们浏览网页时，浏览器作为一个 HTTP 客户端向远端的 HTTP 服务器请求我们想看的网页。HTTP 返回结果后，浏览器再将收到的结果进行渲染，呈现给我们的是漂亮的页面。不过，作为一名技术人员或开发人员，我们想知道的不止于此，而是想透过浏览器，看看隐藏在浏览器背后的东西。因此，在这里我们先通过一个简单的例子来看一下 HTTP 协议。

CURL [1]是一个 HTTP 的命令行工具和客户端，相当于一个命令行版的浏览器。命令行工具与图形界面的工具比起来，前者使用起来更灵活，并且通常更容易展现工具“背后”的东西。CURL 工具使用 -v 命令可以看到 HTTP 协议的通信情况[2]，它与 HTTP 服务器的交互流程如下：

```
$ curl -v http://www.freeswitch.org.cn
* About to connect() to www.freeswitch.org.cn port 80 (#0)
*   Trying 207.97.227.245...
* connected
* Connected to www.freeswitch.org.cn (207.97.227.245) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: www.freeswitch.org.cn
> Accept: */*
```

[1] http://zh.wikipedia.org/zh-cn/CURL。

[2] 使用浏览器的交互过程也类似，只不过是不容易看到这里的交互过程，因此这里我们以 curl 为例。

```
10 >
11 <HTTP/1.1 200 OK
12 < Server: GitHub.com
13 < Date: Sat, 22 Jun 2013 02:30:25 GMT
14 < Content-Type: text/html
15 < Connection: keep-alive
16 < Content-Length: 8685
17 < Last-Modified: Thu, 20 Jun 2013 15:52:14 GMT
18 < Expires: Sat, 22 Jun 2013 02:40:25 GMT
19 < Cache-Control: max-age=600
20 < Vary: Accept-Encoding
21 < Accept-Ranges: bytes
22 < Vary: Accept-Encoding
23 <
24 <html>
25 <head>
26     <meta http-equiv="Content-type" content="text/html; charset=utf-8">
27     <title>FreeSWITCH-CN 中文社区 </title>
28     .... 以下略 ...
```

在上面的例子中，我们用CURL请求网站www.freeswitch.org.cn的首页。其中，以“*”开头的都是CURL的调试信息，以“>”开头的行是客户端向服务器发出的请求，以“<”开头的行是服务器对客户端的响应[1]。

第2行，CURL准备连接网站www.freeswitch.org.cn的80端口；第3行，通过DNS解析得到了与该网站对应的IP地址，并向该地址发起TCP连接请求（HTTP是基于TCP的）；第4和第5行表示TCP连接成功。

连接成功后，客户端在第6行发送一个获取网页的请求：GET / HTTP/1.1。其中，GET表示获取一个资源；“/”是一个路径，表示资源的地址，这里表示网站的根目录，即网站的首页；“HTTP/1.1”为HTTP协议的版本号。User-Agent说明它是一个HTTP客户端用户代理（UA），相当于客户端自己的名片。Accept字段表示接收回复信息的类型，具体的类型称为MIME[2]，常见的有“text/plain”(纯文本)、“text/html”(HTML网页）等。

服务器收到请求后，查找相关资源并返回结果。其中，以“<”开头的行都称为HTTP头（Header)，后续的行称为HTTP正文（Body)。结果是从第11行开始的。“HTTP/1.1 200 OK”也分为三个部分：其中第一部分是协议版本号；200表示状态码；OK是一个可读的描述字符串（称为Reason Phrase)。HTTP的状态码[3]由三位数字构成，说明如下：

- 1xx表示临时响应；
- 2xx表示成功的响应；
- 3xx表示重定向（一般情况下浏览器会对3xx返回的地址重新发起请求，以获取新的资源，称为follow);
- 4xx表示客户端引起的错误（如请求一个不存在的地址时就会收到著名的404）或需

[1] 注意，这里的“*”、“>”及“<”都不是HTTP的一部分，在这里只是为了对各种不同的文本加以区别。

[2] 参见http://zh.wikipedia.org/wiki/多用途互联网邮件扩展。

[3] http://zh.wikipedia.org/wiki/HTTP状态码。

要客户端认证（如 401）；

❑ 5xx 表示服务器端的错误（服务器脚本出错等）。

在本章后面，读者会看到 SIP 协议也有类似的状态码。

所以，这时的“200 OK”表示返回的是一个正常的结果。起始行后面从第 12 行开始跟的是一些 HTTP 头，本例中的 Content-Type 表示返回的资源的类型，它的值“text/html”说明这是一个 HTML 文档。Content-Length 表式文档正文（Body）的长度。在所有的 HTTP 头域结束后，会有一个空行，它标志 HTTP 头部的结束及消息正文的开始。HTTP 响应的正文是从第 24 行开始的。至此，读者应该已经看到熟悉的 HTTP <html> 和 <head> 标签了。如果是浏览器收到这些响应头和正文，就会根据里面的参数对正文进行正确的渲染，呈现给用户漂亮的网页。

Body 是可选的，并不是所有的请求或响应中都有 Body 字段，如果 Content-Length 为 0 或不存在，就表示没有 Body。

SIP 协议也与 HTTP 协议类似，如果你熟悉其中的一种，可以对比着进行学习。我们将从下节开始专门讨论 SIP。

7.1.2 SIP 的基本概念和相关元素

言归正传，我们回到 SIP。SIP 是一个对等的协议，类似 P2P。它和 HTTP 不一样，其不是客户端服务器结构的[⊖]；也不像传统电话那样必须有一个中心的交换机，它可以在不需要服务器的情况下进行通信，只要通信双方都彼此知道对方地址（或者只有一方知道另一方的地址）即可，这种情况称为点对点通信。如图 7-1 所示，Bob 给 Alice 发送一个 INVITE 请求，说“Hi, 一起吃饭吧…”，Alice 说“好的，OK”，电话就接通了。

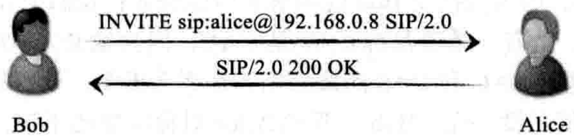


图 7-1 SIP 点对点通信

在 SIP 网络中，Alice 和 Bob 都称为用户代理（User Agent，UA）。UA 是在 SIP 网络中发起或响应 SIP 处理的逻辑实体。UA 是有状态的，也就是说，它维护会话（或称对话）的状态。UA 有两种：一种是 UAC（UA Client），它是发起 SIP 请求的一方，比如图 7-1 中的 Bob；另一种是 UAS（UA Server），它是接受请求并发送响应的一方，比如图 7-1 中的 Alice。由于 SIP 是对等的，当 Alice 呼叫 Bob 时，Alice 就称为 UAC，而 Bob 则实现 UAS 的功能。一般来说，UA 都会实现上述两种功能。

设想 Bob 和 Alice 是经人介绍认识的，而他们还不熟悉，Bob 想请 Alice 吃饭就需要一

⊖ 客户端能请求服务器资源，但服务器不能反过来请求客户端，而 SIP 双方都可以互发请求，即每一个 UA 都可以互做服务器和客户端，下面会讲到。

个中间人（M）传话，而这个中间人就叫代理服务器（Proxy Server）。还有另一种中间人称为重定向服务器（Redirect Server），它以类似于这样的方式工作——中间人 M 告诉 Bob，我也不知道 Alice 在哪里，但我爱人知道，要不然我告诉你我爱人的电话，你直接问她吧，我爱人叫 W。这样，M 就成了一个重定向服务器（把 Bob 对他的请求重定向到他的爱人，这样 Bob 接下来要直接联系他的爱人），而他的爱人 W 是真正的代理服务器。这两种服务器都是 UAS，它们主要是提供一对欲通话的 UA 之间的路由选择功能。

还有一种 UAS 称为注册服务器。试想这样一种情况：Alice 还是个学生，没有自己的手机，但它又希望 Bob 能随时找到她，于是当她在学校时就告诉中间人 M 说她在学校，如果有事找她可以打宿舍的固定电话；如果她要回家，也通知 M 说有事打家里电话；或许某一天她要去姥姥家，也要把她姥姥家的电话告诉 M。总之，只要 Alice 换一个新的位置，它就要向 M 重新“注册”，以让 M 能随时找到她，这时候 M 就相当于一个注册服务器。

还有一种特殊的 UA 称为背靠背用户代理（Back-to-Back UA，B2BUA）。需要指出，其实 RFC 3261 并没有定义 B2BUA 的功能，它只是一对 UAS 和 UAC 的串联。FreeSWITCH 就是一个典型的 B2BUA，事实上，B2BUA 的概念会贯穿本书始终，所以在此我们需要多花一点笔墨来解释。

为了理解 B2BUA，我们来看上述故事的另一个版本。M 和 W 是一对恩爱夫妻。M 认识 Bob 而 W 认识 Alice。M 和 W 有意撮合两个年轻人，但见面时由于两人太腼腆而互相没留电话号码。事后 Bob 想知道 Alice 对他感觉如何，于是打电话问 M，M 不认识 Alice，就转身问爱人 W（注意这次 M 没有直接把 W 的电话给 Bob），W 紧接着打电话给 Alice，Alice 说印象还不错，W 就把这句话告诉 M，M 又转过身告诉 Bob。这样，M 和 W 一个面向 Bob，一个对着 Alice，他们两个合在一起，称为 B2BUA。在这里，Bob 是 UAC，因为他发起请求；M 是 UAS，因为他接受 Bob 的请求并为他服务；我们把 M 和 W 看做一个整体，他们背靠着背（站着、坐着、躺着都行），W 是 UAC，因为她又向 Alice 发起了请求，最后 Alice 是 UAS。其实这里 UAC 和 UAS 的概念也不是那么重要，重要的是要理解这个**背靠背的用户代理**。因为事情还没有完，Bob 一听说 Alice 对他印象还不错，开心得不得了，便想请抽空请 Alice 吃饭，他将这一想法告诉 M，M 告诉 W，W 又告诉 Alice。然后 Alice 问去哪里吃啊，W 又只好问 M，M 再问 Bob…… 在这对年轻人挂断电话之前，M 和 W 只能“背对背”不停地工作，如图 7-2 所示。

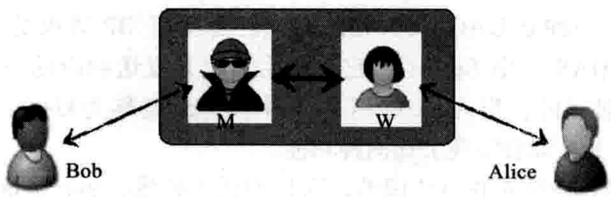


图 7-2 B2BUA

从图 7-2 可以看出，四个人其实全是 UA。当然，虽然 FreeSWITCH 是 B2BUA，但也

可以经过特殊的配置，实现一些代理服务器和重定向服务器的功能，甚至也可以从中间劈开，两边分别作为一个普通的 UA 来工作。这没有什么奇怪的，在 SIP 世界中，所有 UA 都是平等的。具体到实物，则 M 和 W 就组成了实现软交换功能的交换机，它们对外说的语言是 SIP，而在内部它们使用自己家的语言沟通。Bob 和 Alice 就分别成了我们常见的软电话，或者硬件的 SIP 话机。

此外，还有一个概念，称为边界会话控制器（Session Border Controller，SBC）。它主要位于一堆 SIP 服务器的边界，用于隐藏内部服务器的拓扑结构、抵御外来攻击等。SBC 可能是一个代理服务器，也可能是一个 B2BUA。其应用位置和拓扑结构如图 7-3 所示。

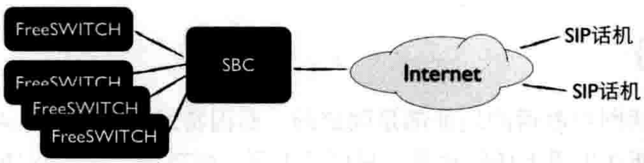


图 7-3 SBC 示意图

一般来说，SBC 具有两个或多个网络接口卡（网卡），一边连接互联网，一边连接内部的网络。从互联网上对内部 SIP 服务器的访问只能经过 SBC，因而这种拓扑结构能从一定程序上保证内部 SIP 服务器的安全。其他的部署模式还包括更高级别的防火墙设备，以及把 SBC 设备放在网络拓扑结构的防火区（DMZ）中等，以最大限度保证安全。

7.1.3 SIP 协议的基本方法和头域简介

SIP 定义了 6 种基本方法，如表 7-1 所示。

表 7-1 SIP 的 6 种基本方法

基本方法	说 明
REGISTER	注册联系信息
INVITE	初始化一个会话，可以理解为发起一个呼叫
ACK	对 INVITE 消息的最终响应
CANCEL	取消一个等待处理或正在处理的请求
BYE	终止一个会话
OPTIONS	查询服务器和能力，也可以用作 ping 测试

除此之外，SIP 还定义了一些扩展方法，如 SUBSCRIBE、NOTIFY、MESSAGE、REFER、INFO 等，等到我们用到时会简单介绍。

另外，无论是基本方法还是扩展方法，所有 SIP 消息都必须包含以下 6 个头域，如表 7-2 所示。

在后面的例子中，我们会看到这些方法和头域在实际的 SIP 消息中是如何应用的，在此就不多着笔墨了。

表 7-2 SIP 必须包含的头域

名 称	描 述
Call-ID	用于区分不同会话的唯一标志
CSeq	顺序号，用于在同一会话中区分事务
From	说明请求来源
To	说明请求接受方
Max-Forwards	限制跳跃点数和最大转发次数
Via	描述请求消息经过的路径

7.2 SIP 注册

普通的固定电话网中电话的地址都是固定的，而因特网是开放的。Alice 可能在家也可能在学校，甚至可能在世界上任何角落，只要能上网，她就能与全世界通信。当然，如果她作为被叫的一方，为了让我们的 FreeSWITCH 服务器能随时找到她，她的 UA 必须向我们的服务器**注册**。

通常的注册流程是，Alice 向 FreeSWITCH 发起注册（REGISTER）请求，FreeSWITCH 返回 401 消息对 Alice 发起 Challenge（挑战），Alice 将自己的用户名密码信息与收到的 Challenge 信息进行计算，并将计算结果以加密的形式附加到下一个 REGISTER 请求上，重新发起注册，FreeSWITCH 收到后对本地数据库中保存的 Alice 的信息使用同样的算法进行计算和加密，并将其与 Alice 发过来的计算结果相比较。如果计算结果相匹配，则认证通过，Alice 便可以正常注册。交互流程如图 7-4 所示。

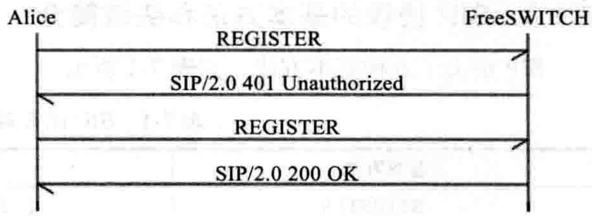


图 7-4 SIP 注册流程

下面我们用真正的注册流程进行说明。下面的 SIP 消息是在真正的 FreeSWITCH 中跟踪（trace）出来的。其中 FreeSWITCH 服务器的 IP 地址是 192.168.4.4，它使用默认的端口号 5060，在这里我们使用的 SIP 承载方式是 UDP。Alice 使用的 UA 是 Zoiper，端口号是 5090（在笔者写作时它与 FreeSWITCH 在同一台机器上，所以不能再使用端口 5060）。其中，每个消息短横线之间的内容都是 FreeSWITCH 中输出的调试信息，不是 SIP 的一部分。recv（Receive 的缩写形式）表示 FreeSWITCH 收到的消息，send 表示发出的消息，下同。

在 Alice 发起注册时，下面是 FreeSWITCH 收到的第一条消息（为了便于说明，在排版时增加了行号）：

```
------------------------------------------------------------------------
recv 584 bytes from udp/[192.168.4.4]:5090 at 12:30:57.916812:
------------------------------------------------------------------------
01 REGISTER sip:192.168.4.4;transport=UDP SIP/2.0
```

```
02 Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d9ed3bbae47e568b-1-
--d8754z-;rport
03 Max-Forwards: 70
04 Contact: <sip:Alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>
05 To: <sip:Alice@192.168.4.4;transport=UDP>
06 From: <sip:Alice@192.168.4.4;transport=UDP>;tag=9c709222
07 Call-ID: NmFjNzA3MWY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY2OGE.
08 CSeq: 1 REGISTER
09 Expires: 3600
10 Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
11 User-Agent: Zoiper rev.5415
12 Allow-Events: presence
13 Content-Length: 0
```

我们前面已经说过，SIP 是类似 HTTP 的纯文本的协议，所以很容易阅读。其中：

- 第 1 行的 REGISTER 表示这是一条注册消息。
- 第 2 行的 Via 是 SIP 的消息路由，如果 SIP 经过好多代理服务器转发，则会有多条 Via 记录。
- 第 3 行，Max-forwards 指出消息最多可以经过多少次转发，主要是为了防止产生死循环。
- 第 4 行，Contact 是 Alice 的联系地址，即相当于 Alice 家的地址，本例中 FreeSWITCH 应该能在 192.168.4.4 这台机器上的 5090 端口找到她。
- 第 5 和第 6 行的 To 和 From 表示以 Alice 注册。
- 第 7 行，Call-ID 是本次 SIP 会话（Session）的标志。
- 第 8 行，CSeq 是一个序号，由于 UDP 是不可靠的协议，在不可靠的网络上可能丢包，所以有些包需要重发，该序号则可以防止重发引起的消息重复。
- 第 9 行，Expires 是说明本次注册的有效期，单位是秒。在本例中，Alice 的注册信息会在一小时后失效，它应该在半小时内再次向 FreeSWITCH 注册，以免 FreeSWITCH“忘记”她。实际上，大部分 UA 的实现都会在几十秒内重新发一次注册请求，这在 NAT 的网络中有助于保持连接。
- 第 10 行，Allow 是说明 Alice 的 UA 所能支持的功能，某些 UA 功能丰富，而某些 UA 仅有有限的功能。
- 第 11 行，User-Agent 是 UA 的型号。
- 第 12 行，Allow-Events 说明她允许哪些事件通知。
- 第 13 行，Content-Length 是消息正文（Body）的长度，在这里只有消息头（Header），没有消息正文，因此长度为 0 。

FreeSWITCH 需要验证 Alice 的身份才允许她注册。在 SIP 中，没有发明新的认证方式，而是使用已有的 HTTP 摘要⊖（Digest）方式来认证。这里它给 Alice 回送（发，send）401 响应消息，消息内容如下：

⊖ 参见：http://zh.wikipedia.org/wiki/HTTP 摘要认证。

```
------------------------------------------------------------------------
send 664 bytes to udp/[192.168.4.4]:5090 at 12:30:57.919364:
------------------------------------------------------------------------
SIP/2.0 401 Unauthorized
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d9ed3bbae47e568b-1---
d8754z-;rport=5090
From: <sip:Alice@192.168.4.4;transport=UDP>;tag=9c709222
To: <sip:Alice@192.168.4.4;transport=UDP>;tag=QFXyg6gcByvUH
Call-ID: NmFjNzA3MWY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY2OGE.
CSeq: 1 REGISTER
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
WWW-Authenticate: Digest realm="192.168.4.4",
    nonce="62fb812c-71d2-4a36-93d6-e0008e6a63ee",
algorithm=MD5, qop="auth"
Content-Length: 0
```

401 消息表示未认证，它是 FreeSWITCH 对 Alice 请求的响应。各消息头的含义基本上与请求中的含义是一样的。其中，“CSeq: 1 REGISTER”表示它是针对刚刚收到的 CSeq 为 1 的 REGISTER 请求的响应。同时，它在本端生成一个认证摘要（WWW-Authenticate），一起发送给 Alice。

Alice 收到带有摘要的 401 后，重新发起注册请求，这一次加上了根据收到的摘要和她自己的用户名密码生成的认证信息（Authorization 头）。同时，下面你可能也会注意到，CSeq 序号变成了 2。下面是重发的注册请求信息：

```
------------------------------------------------------------------------
recv 846 bytes from udp/[192.168.4.4]:5090 at 12:30:57.921011:
------------------------------------------------------------------------
REGISTER sip:192.168.4.4;transport=UDP SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-dae1693be9f8c10d-1---
d8754z-;rport
Max-Forwards: 70
Contact: <sip:Alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>
To: <sip:Alice@192.168.4.4;transport=UDP>
From: <sip:Alice@192.168.4.4;transport=UDP>;tag=9c709222
Call-ID: NmFjNzA3MWY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY2OGE.
CSeq: 2 REGISTER
Expires: 3600
Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
User-Agent: Zoiper rev.5415
Authorization: Digest username="Alice",realm="192.168.4.4",
    nonce="62fb812c-71d2-4a36-93d6-e0008e6a63ee",
    uri="sip:192.168.4.4;transport=UDP",
    response="32b5ddaea8647a3becd25cb84346b1c3",
    cnonce="b4c6ac7e57fc76b85df9440994e2ede8",
    nc=00000001,qop=auth,algorithm=MD5
Allow-Events: presence
Content-Length: 0
```

FreeSWITCH 收到带有认证的注册消息后，核实 Alice 身份，如果认证通过，则向 Alice 回应 200 OK 消息，表示注册成功了。返回的 200 OK 消息如下：

```
------------------------------------------------------------------------
send 665 bytes to udp/[192.168.4.4]:5090 at 12:30:57.936940:
------------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-dae1693be9f8c10d-1---
d8754z-;rport=5090
From: <sip:Alice@192.168.4.4;transport=UDP>;tag=9c709222
To: <sip:Alice@192.168.4.4;transport=UDP>;tag=rrpQj11F86jeD
Call-ID: NmFjNzA3MWY1MDI3NGViMjY1N2QwZDlmZWQ5ZGY2OGE.
CSeq: 2 REGISTER
Contact: <sip:Alice@192.168.4.4:5090;rinstance=d42207a765c0626b;transport=UDP>;
expires=3600
Date: Tue, 27 Apr 2010 12:30:57 GMT
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Content-Length: 0
```

如果认证失败（可能是 Alice 的密码填错了），则回应 403 Forbidden 或其他失败消息，消息内容如下：

```
------------------------------------------------------------------------
send 542 bytes to udp/[192.168.4.4]:5090 at 13:22:49.195554:
------------------------------------------------------------------------
SIP/2.0 403 Forbidden
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-d447f43b66912a1b-1---
d8754z-;rport=5090
From: <sip:Alice@192.168.4.4;transport=UDP>;tag=c097e17f
To: <sip:Alice@192.168.4.4;transport=UDP>;tag=yeecX364pvryj
Call-ID: ZjkxMGJmMjE4Y2ZiNjU5MzM5NDZkMTE5NzMzMzM0Mjc.
CSeq: 2 REGISTER
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Content-Length: 0
```

可以看到，在整个注册过程中，Alice 的密码是不会直接在 SIP 消息中传送的，因而最大限度地保证了认证过程的安全。

如果 Alice 注册成功，则 FreeSWITCH 会将 Alice 在 SIP 消息中的联系地址（Contact 字段）记录下来。以后如果有人呼叫 Alice，FreeSWITCH 就可以向该联系地址发送 SIP 消息以建立呼叫。

7.3 SIP 呼叫流程

上一节，我们讨论了 SIP 注册。SIP UA 向 FreeSWITCH 注册的主要是为了在呼叫时 FreeSWITCH 能够找到它。SIP 的主要意义也是“处理会话的发起和终结”，通俗地讲就是处理和控制呼叫。下面我们就来讨论几个典型的 SIP 呼叫流程。

7.3.1 UA 间直接呼叫

前面我们讲过，SIP 的 UA 都是平等的，如果一方知道另一方的地址，就可以通信。我们先来做一个实验。在笔者的机器上，启动了两个软电话（UA），一个是 Bob 的 X-Lite（左），另一个是 Alice 的 Zoiper。它们的 IP 地址都是 192.168.4.4，而端口号分别是 26000 和 5090，当 Bob 呼叫 Alice 时，它只需直接呼叫 Alice 的 SIP 地址 sip:Alice@192.168.4.4:5090 ㊀。如图 7-5 所示，Alice 的电话正在振铃。



图 7-5 SIP UA 间直接呼叫

详细的呼叫流程如图 7-6 所示。

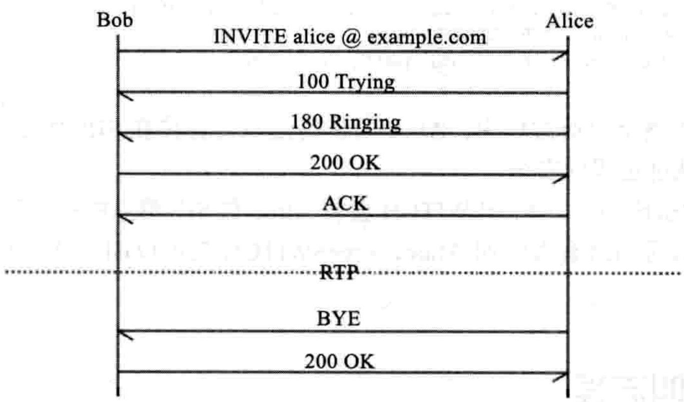


图 7-6 SIP UA 间直接呼叫流程图

㊀ 注意，在最新版的 X-Lite 上（如 X-Lite 4.5.4），不允许直接使用 SIP 地址进行呼叫，解决办法是添加一个 SIP 账号，其中 Domain 中填写对方的 SIP 地址，如本例中的 192.168.4.4:5090。其他参数可以随便填，最后记住去掉“Register with domain and receive calls”前面的复选框的选中状态。保存后，就可以呼叫 Alice 了，这跟上述流程一样。

首先 Bob 向 Alice 发送 INVITE 消息请求建立 SIP 会话。Alice 的 UA 回 100 Trying 消息，意思是说我收到你的请求了，先等一会儿。接着 Alice 的电话开始振铃，并给对方回消息 180 Ringing，说我这边已经振铃了，Alice 听到后可能一会就过来接电话。Bob 的 UA 收到该消息后即可以播放回铃音，以提示 Bob 对方的话机正在振铃。接着 Alice 接了电话，她发送 200 OK 消息给 Bob，该消息是对 INVITE 消息的最终响应（所有大于 1 的状态码都是最终响应），而先前的 100 和 180 消息都是临时状态，只是表明呼叫进展的情况。Bob 收到 200 后向 Alice 回复 ACK 证实消息。INVITE - 200 - ACK 完成“三次握手”的操作，保证了呼叫可以正常进行。其中，INVITE - 1xx - 200 等消息合在一起称为一个事务（Transaction）。这时候 Bob 已经在跟 Alice 通话了，他们通话的内容（语音数据）是在 SIP 之外的 RTP 包中传递的，我们后面再详细讨论 RTP。

最后，Alice 挂断电话，向 Bob 发送 BYE 消息，Bob 收到 BYE 后回送 200 OK，通话完毕。其中 BYE 和 200 OK 也是一个事务，而上面的所有消息，则称为一个对话（Dialog，也译作会话）。

反过来也一样，Alice 可以直接呼叫 Bob 的地址 sip:Bob@192.168.4.4:26000 进行通话。

上面描述了一个最简单的 SIP 呼叫流程。实际上，SIP 还有其他一些消息。SIP 消息大致可分为请求和响应两类。请求由 UAC 发出，到达 UAS 后，UAS 回送响应消息。某些响应消息需要证实（ACK），以完成三次握手。其中请求消息包括基本的 INVITE、ACK、OPTIOS、BYE、CANCEL、REGISTER，以及 re-INVITE、PRACK、SUBSCRIBE、NOTIFY、UPDATE、MESSAGE、REFER 等一些扩展。而响应消息则都包含一个状态码和一个原因短语（Reason Phrase）。与 HTTP 响应类似，状态码由三位数字组成：

- 1xx 组的响应为临时状态，表明呼叫进展的情况；
- 2xx 表明请求已被成功收到、理解和接受；
- 3xx 为重定向，表明 SIP 请求需要转向到另一个 UAS 处理；
- 4xx 表明请求失败，这种失败一般是由客户端或网络引起的，如密码错误、空号等，客户端应该重新修改请求，然后重发；
- 5xx 为服务器内部错误，表明服务器出错，不能响应合法的请求；
- 6xx 为全局性错误，如 600 Busy Everywhere。

状态码后面跟着一个原因短语（如 200 OK 中的 OK 及刚才讲到的 Busy Everywhere），它是对前面的状态码的一个简单解释。

7.3.2　通过 B2BUA 呼叫

在真实世界中，Bob 和 Alice 可以会经常改变位置，那么它们的 SIP 地址也会相应改变，并且，如果他们之中有一个或两个处于 NAT 的网络中时，直接通信就更困难了。所以，他们通常会借助于一个服务器来实现通信。这样 Bob 和 Alice 通过注册到服务器上即可获得一个服务器上的公有 SIP 地址。注册服务器的地址一般是不变的，因此他们的 SIP 地址就不会发生变化，他们也就总是能够进行通信了。

现在，作为例子，我们让他们两个都注册到 FreeSWITCH 上。上面已经说过，FreeSWITCH 监听的端口是 SIP 默认的 5060 端口。Bob 和 Alice 注册后，他们分别获得了一个服务器的地址（SIP URI）：sip:Bob@192.168.4.4 和 sip:Alice@192.168.4.4（默认的端口号 5060 可以省略）。

下面是 Bob 呼叫 Alice 的流程。需要指出，如果 Bob 只是发起呼叫而不接收呼叫，他并不需要向 FreeSWITCH 注册（有些软交换服务器规定需要先注册才能发起呼叫，但 SIP 是不强制这么做的）。

首先，Bob 向 FreeSWITCH 发送 INVITE 消息，请求建立一个呼叫，发送的 INVITE 消息如下：

```
------------------------------------------------------------------------
recv 1118 bytes from udp/[192.168.4.4]:26000 at 13:31:39.938891:
------------------------------------------------------------------------
INVITE sip:Alice@192.168.4.4 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---
d8754z-;rport
Max-Forwards: 70
Contact: <sip:Bob@192.168.4.4:26000>
To: "Alice"<sip:Alice@192.168.4.4>
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 1 INVITE
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
Content-Type: application/sdp
User-Agent: X-Lite release 1014k stamp 47051
Content-Length: 594
```

上面的消息中省略了 SDP 的内容，关于 SDP 的部分我们将留到本章后面再探讨。在此，Bob 的 UAC 通过 INVITE 消息向 FreeSWITCH 发起请求。Bob 的 UAC 用的是 X-Lite（User-Agent），它运行在端口 26000 上（由 Contact 字段给出。实际上，它默认的端口也是 5060，但由于在笔者的实验环境下它与 FreeSWITCH 运行在一台机器上，5060 端口已被 FreeSWITCH 占用，因此需要选择另一个端口）。其中，From 为主叫用户的地址，To 为被叫用户的地址。此时 FreeSWITCH 作为一个 UAS 接受请求并进行响应。但在此之前，它先通过 100 Trying 消息通知 Bob 它已经收到了他的请求，消息内容如下：

```
------------------------------------------------------------------------
send 345 bytes to udp/[192.168.4.4]:26000 at 13:31:39.940278:
------------------------------------------------------------------------
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---
d8754z-;rport=26000
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
To: "Alice"<sip:Alice@192.168.4.4>
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 1 INVITE
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Content-Length: 0
```

FreeSWITCH 通过 100 Trying 消息告诉 Bob：“我已经收到你的消息了，别着急，我正

在设法联系 Alice……"，该消息称为呼叫进展消息。

但就在此时，FreeSWITCH 发现它还不认识 Bob，即它还不确定 Bob 是否有权通过它发起呼叫，因而它需要确认 Bob 的身份。在 SIP 中，它通过回送带有 Digest 验证信息（Proxy-Authenticate 头）的 407 消息来通知 Bob（类似于注册流程中的 401），消息内容如下：

```
------------------------------------------------------------------------
send 826 bytes to udp/[192.168.4.4]:26000 at 13:31:39.943392:
------------------------------------------------------------------------
SIP/2.0 407 Proxy Authentication Required
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---
d8754z-;rport=26000
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
To: "Alice" <sip:Alice@192.168.4.4>;tag=B4pem31jHgtHS
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 1 INVITE
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Accept: application/sdp
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
    include-session-description, presence.winfo, message-summary, refer
Proxy-Authenticate: Digest realm="192.168.4.4",
    nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8",
    algorithm=MD5, qop="auth"
Content-Length: 0
```

Bob 回送 ACK 证实消息向 FreeSWITCH 证实已收到认证要求，至此，从 INVITE 开始发起的事务就结束了。下面是 FreeSWITCH 收到的 ACK 消息：

```
------------------------------------------------------------------------
recv 319 bytes from udp/[192.168.4.4]:26000 at 13:31:39.945314:
------------------------------------------------------------------------
ACK sip:Alice@192.168.4.4 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-56adad736231f024-1---
d8754z-;rport
To: "Alice" <sip:Alice@192.168.4.4>;tag=B4pem31jHgtHS
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 1 ACK
Content-Length: 0
```

然后，Bob 重新发送 INVITE 请求。这次还附带了 Proxy-Authorization 验证信息，该信息是根据上次收到的 Proxy-Authenticate 中的信息与 Bob 提供的认证用户名和密码计算出的结果。该 INVITE 重新开始了一个事务，消息内容如下：

```
------------------------------------------------------------------------
recv 1376 bytes from udp/[192.168.4.4]:26000 at 13:31:39.945526:
------------------------------------------------------------------------
INVITE sip:Alice@192.168.4.4 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---
d8754z-;rport
Max-Forwards: 70
```

```
    Contact: <sip:Bob@192.168.4.4:26000>
    To: "Alice"<sip:Alice@192.168.4.4>
    From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
    Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
    CSeq: 2 INVITE
    Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
    Content-Type: application/sdp
    Proxy-Authorization: Digest username="Bob",realm="192.168.4.4",
        nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8",
        uri="sip:Alice@192.168.4.4",response="327887635344405bcd545da06763c466",
        cnonce="c164b74f625ff2161bd8d47dba3a0ee2",
        nc=00000001,qop=auth,
        algorithm=MD5
    User-Agent: X-Lite release 1014k stamp 47051
    Content-Length: 594
```

这里也省略了 SDP 消息正文。

FreeSWITCH 收到 INVITE 请求后，重新回送 100 Trying，通知 Bob 呼叫进展情况，消息内容如下：

```
    ------------------------------------------------------------------------
    send 345 bytes to udp/[192.168.4.4]:26000 at 13:31:39.946349:
    ------------------------------------------------------------------------
    SIP/2.0 100 Trying
    Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---
d8754z-;rport=26000
    From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
    To: "Alice"<sip:Alice@192.168.4.4>
    Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
    CSeq: 2 INVITE
    User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
    Content-Length: 0
```

然后 FreeSWITCH 对 Bob 发来的认证信息进行验证，发现 Bob 是本地的授权用户，因而呼叫可以继续进行。

至此，Bob 与 FreeSWITCH 之间的通信已经初步建立，这种通信的逻辑通道称为一个 Channel。该 Channel 是由 Bob 的 UA 和 FreeSWITCH 的一个 UA 构成的，我们称之为 FreeSWITCH 的一条腿，又称 a-leg。

接下来，呼叫进入路由阶段，FreeSWITCH 根据路由，发现 Bob 要呼叫 Alice，由于 FreeSWITCH 是一个 B2BUA，因而它需要要建立另外一条腿去呼叫 Alice，与 a-leg 相对，这条新腿就称为 b-leg。

FreeSWITCH 通过查找本地数据库，得到了 Alice 的位置（Alice 的联系地址，Alice 必须事先已经向它注册），接着启动一个 UA（用作 UAC），向 Alice 发送一个新的 INVITE 消息，内容如下（注意，这完全是一个新的事务，理论上与 a-leg 没有任何关系）：

```
    ------------------------------------------------------------------------
    send 1340 bytes to udp/[192.168.4.4]:5090 at 13:31:40.028988:
    ------------------------------------------------------------------------
    INVITE sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP
SIP/2.0
```

```
Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKey90QUyHZQXNN
Route: <sip:Alice@192.168.4.4:5090>;rinstance=e7d5364c81f2b879;transport=UDP
Max-Forwards: 69
From: "Bob" <sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
To: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 130069214 INVITE
Contact: <sip:mod_sofia@192.168.4.4:5060>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
    include-session-description, presence.winfo, message-summary, refer
Content-Type: application/sdp
Content-Disposition: session
Content-Length: 313
X-FS-Support: update_display
Remote-Party-ID: "Bob" <sip:Bob@192.168.4.4>;party=calling;screen=yes;privacy=off
```

可以看到，该 INVITE 的 Call-ID 与上一次 INVITE 消息中的不同，说明这是另一个 SIP 对话（Dialogue）。另外，消息中还多了一个 Remote-Party-ID，它主要是用来支持主叫号码显示功能（俗称来电显示，有了这个功能，Alice 的话机上就可以显示 Bob 的号码，知道是 Bob 打来的）。与普通的 POTS 通话不同，在 SIP 通话话中，不仅能显示电话号码（这里是 Bob），还能显示一个可选的名字（Bob）。这也说明了 FreeSWITCH 这个 B2BUA 本身是一个整体，它虽然是以一个单独的 UA 呼叫 Alice，但还是跟负责 Bob 的那个 UA 有联系——就是这种背靠背的串联关系，使得 Alice 这一侧的 Channel（b-leg）知道 Bob 那一侧（a-leg）的一些信息（如这里的主叫号码）。

同理，Alice 会回送 100 Trying 消息通知 FreeSWITCH 它已经正常接收到该 INVITE 请求，正准备进行下一步处理。消息内容如下：

```
------------------------------------------------------------------------
recv 309 bytes from udp/[192.168.4.4]:5090 at 13:31:40.193634:
------------------------------------------------------------------------
SIP/2.0 100 Trying
Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
To: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
From: "Bob" <sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 130069214 INVITE
Content-Length: 0
```

虽然我们曾经说过，所有的 SIP UA 都是对等的。但 Alice 的身份与 FreeSWITCH 不同，它作为一个被叫用户，不需要对 FreeSWITCH 发起的呼叫进行认证，因而，Alice 的话机直接振铃，并向 FreeSWITCH 回送 180 Ringing 消息，通知 FreeSWITCH Alice 的话机已经开始振铃了。FreeSWITCH 收到的 180 消息如下：

```
------------------------------------------------------------------------
recv 431 bytes from udp/[192.168.4.4]:5090 at 13:31:40.193816:
------------------------------------------------------------------------
```

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
Contact: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
To: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
From: "Bob"<sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 130069214 INVITE
User-Agent: Zoiper rev.5415
Content-Length: 0
```

上述代码中，180 也是呼叫进展消息，它说明 Alice 的电话已经开始振铃了，如果 Alice 听到了，则过一会她可能会接听电话。FreeSWITCH 在收到 180 以后，有以下两个选择。

- 直接给 Bob 回 180 消息。Bob 的话机收到 180 后，知道 Alice 那边已经振铃了。注意，这里只是 Bob 的话机知道，Bob 并不知道收到了 180。因而 Bob 的话机会自己产生一个回铃音，播放给 Bob 听，以提示 Bob 对方正在振铃。
- 给 Bob 回 183 消息。与 180 不同，183 消息包含媒体 SDP。这时，FreeSWITCH 就有机会产生一个回铃音，通过 RTP 发送给 Bob。Bob 的话机就不需要自己再产生回铃音。通过这种技术，FreeSWITCH 可以为呼叫 Alice 的用户产生个性化的回铃音，也称彩铃。

不过，在此我们先不深入讨论。不管是 180 还是 183，Bob 都会听到回铃音。FreeSWITCH 默认是 183，消息内容如下：

```
send 1125 bytes to udp/[192.168.4.4]:26000 at 13:31:40.270533:
------------------------------------------------------------------------
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---
d8754z-;rport=26000
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
To: "Alice" <sip:Alice@192.168.4.4>;tag=cDg7NyjpeSg4m
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 2 INVITE
Contact: <sip:Alice@192.168.4.4:5060;transport=udp>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Accept: application/sdp
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
    include-session-description, presence.winfo, message-summary, refer
Content-Type: application/sdp
Content-Disposition: session
Content-Length: 267
Remote-Party-ID: "Alice" <sip:Alice@192.168.4.4>
```

这里省略了 SDP 消息。

再看 Alice 这端，她听到电话响了，一看主叫号码是 Bob，心里非常甜蜜，便迫不及待地接起电话。在她接起电话的一刹那，她的 SIP 话机给 FreeSWITCH 回送了 200 OK 消息，表示 Alice 已经接听了。消息内容如下：

```
--------------------------------------------------------------------------------
recv 768 bytes from udp/[192.168.4.4]:5090 at 13:31:43.251980:
--------------------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKey90QUyHZQXNN
Contact: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
To: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=
3813e926
From: "Bob"<sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 130069214 INVITE
Allow: INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO,
SUBSCRIBE
Content-Type: application/sdp
User-Agent: Zoiper rev.5415
Content-Length: 226
```

这里省略了SDP消息。

FreeSWITCH向Alice回送ACK证实消息，证实它已经收到200 OK了。至此，FreeSWITCH与Alice这一端的通话已建立完毕。ACK消息内容如下：

```
--------------------------------------------------------------------------------
send 436 bytes to udp/[192.168.4.4]:5090 at 13:31:43.256692:
--------------------------------------------------------------------------------
ACK sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKF72SSpFNv0K8g
Max-Forwards: 70
From: "Bob" <sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
To: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;tag=3813e926
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 130069214 ACK
Contact: <sip:mod_sofia@192.168.4.4:5060>
Content-Length: 0
```

当然，FreeSWITCH作为中间人也不敢怠慢，它立即向Bob的话机发送200 OK消息，并切断原先自动产生的回铃音，把Alice甜美的声音接进来。200 OK消息如下：

```
--------------------------------------------------------------------------------
send 1135 bytes to udp/[192.168.4.4]:26000 at 13:31:43.293311:
--------------------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-87d60b47b6627c3a-1---
d8754z-;rport=26000
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
To: "Alice" <sip:Alice@192.168.4.4>;tag=cDg7NyjpeSg4m
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 2 INVITE
Contact: <sip:Alice@192.168.4.4:5060;transport=udp>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Allow-Events: talk, presence, dialog, line-seize, call-info, sla,
    include-session-description, presence.winfo, message-summary, refer
Session-Expires: 120;refresher=uas
```

```
Min-SE: 120
Content-Type: application/sdp
Content-Disposition: session
Content-Length: 267
Remote-Party-ID: "Alice" <sip:Alice@192.168.4.4>
```

这里省略了 SDP 消息。

多半这时 Alice 已经开始说话了："Hi, Bob，你好……"，Bob 的 UA 在收到 200 消息后也启动本地的麦克风，至此，他们两个就可以对话了。当然，Bob 作为一个有责任的男人，他还顺便通过 ACK 消息通知 FreeSWITCH 他确实收到了上面的 200 消息。ACK 消息如下：

```
------------------------------------------------------------------------
recv 697 bytes from udp/[192.168.4.4]:26000 at 13:31:43.413025:
------------------------------------------------------------------------
ACK sip:Alice@192.168.4.4:5060;transport=udp SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:26000;branch=z9hG4bK-d8754z-ef53864320037c04-1---
d8754z-;rport
Max-Forwards: 70
Contact: <sip:Bob@192.168.4.4:26000>
To: "Alice"<sip:Alice@192.168.4.4>;tag=cDg7NyjpeSg4m
From: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 2 ACK
Proxy-Authorization: Digest username="Bob",realm="192.168.4.4",
    nonce="31c5c3e0-cc6e-46c8-a661-599b0c7f87d8",
    uri="sip:Alice@192.168.4.4",response="327887635344405bcd545da06763c466",
    cnonce="c164b74f625ff2161bd8d47dba3a0ee2",nc=00000001,qop=auth,
    algorithm=MD5
User-Agent: X-Lite release 1014k stamp 47051
Content-Length: 0
```

至此，a-leg 和 b-leg 的通话分别建立完毕，FreeSWITCH 作为一个中间人将两个 leg 桥接起来，使得双方可以正常通话，通话进入稳定阶段。

一般来说，在需要计费的场合，FreeSWITCH 可以对 Bob 进行计费。计费的开始时间是以 FreeSWITCH 给 Bob 发送 200 的时间为准的。当然，FreeSWITCH 本身并不做任何计费处理，它只是记录通话的起止时间，实际的费率和计费工作由其他软件完成。

通话进入稳定阶段后，一般不再有 SIP 消息交互，所有的语音数据都在 RTP 中传送。

"电话粥"也不能煲太久，终于 Alice 挂断了电话，它的话机会给 FreeSWITCH 发送 BYE 消息，通知 FreeSWITCH 要再见（拆线）了。消息内容如下：

```
------------------------------------------------------------------------
recv 484 bytes from udp/[192.168.4.4]:5090 at 13:31:49.949240:
------------------------------------------------------------------------
BYE sip:mod_sofia@192.168.4.4:5060 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-2146ae0ddd113efe-1---
d8754z-;rport
Max-Forwards: 70
Contact: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>
To: "Bob"<sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
From: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;ta
        g=3813e926
```

```
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 2 BYE
User-Agent: Zoiper rev.5415
Content-Length: 0
```

FreeSWITCH 回送 200 OK，并释放 b-leg。消息内容如下：

```
----------------------------------------------------------------------------
send 543 bytes to udp/[192.168.4.4]:5090 at 13:31:49.950425:
----------------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.4:5090;branch=z9hG4bK-d8754z-2146ae0ddd113efe-1---
d8754z-;rport=5090
From: <sip:Alice@192.168.4.4:5090;rinstance=e7d5364c81f2b879;transport=UDP>;ta
        g=3813e926
To: "Bob"<sip:Bob@192.168.4.4>;tag=Dp9ZQS3SB26pg
Call-ID: 0d74ac35-cca4-122d-81a2-2990e5b2bd3e
CSeq: 2 BYE
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Content-Length: 0
```

同时，FreeSWITCH 也给 Bob 发送 BYE 消息，通知 Bob 要拆线了。在下面的消息中，它还包含了挂机原因（Hangup Cause，在 Reason 头中给出)，此处 NOMAL_CLEARING 表示正常释放。BYE 消息内容如下：

```
----------------------------------------------------------------------------
send 630 bytes to udp/[192.168.4.4]:26000 at 13:31:50.003165:
----------------------------------------------------------------------------
BYE sip:Bob@192.168.4.4:26000 SIP/2.0
Via: SIP/2.0/UDP 192.168.4.4;rport;branch=z9hG4bKggvjUH0rS99tc
Max-Forwards: 70
From: "Alice" <sip:Alice@192.168.4.4>;tag=cDg7NyjpeSg4m
To: "Bob" <sip:Bob@192.168.4.4>;tag=15c8325a
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 130069219 BYE
Contact: <sip:Alice@192.168.4.4:5060;transport=udp>
User-Agent: FreeSWITCH-mod_sofia/1.0.trunk-16981M
Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, UPDATE, INFO, REGISTER, REFER,
    NOTIFY, PUBLISH, SUBSCRIBE
Supported: timer, precondition, path, replaces
Reason: Q.850;cause=16;text="NORMAL_CLEARING"
Content-Length: 0
```

Bob 的话机回送 200 OK 消息，FreeSWITCH 收到后，也释放 a-leg，通话结束。FreeSWITCH 收到的 200 OK 消息如下：

```
----------------------------------------------------------------------------
recv 367 bytes from udp/[192.168.4.4]:26000 at 13:31:50.111765:
----------------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.4.4;rport=5060;branch=z9hG4bKggvjUH0rS99tc
Contact: <sip:Bob@192.168.4.4:26000>
```

```
To: "Bob"<sip:Bob@192.168.4.4>;tag=15c8325a
From: "Alice"<sip:Alice@192.168.4.4>;tag=cDg7NyjpeSg4m
Call-ID: YWEwYjNlZTZjOWZjNDg3ZjU3MjQ3MTA1ZmQ1MDM5YmQ.
CSeq: 130069219 BYE
User-Agent: X-Lite release 1014k stamp 47051
Content-Length: 0
```

上面描述的是一个典型的通过 FreeSWITCH 进行呼叫的流程。虽然有点长，但是并不复杂，图 7-7 所示可以很形象地描述出了 FreeSWITCH 的两条“腿”——a-leg 和 b-leg。



图 7-7 B2BUA 的两条腿

完整的呼叫流程如图 7-8 所示。

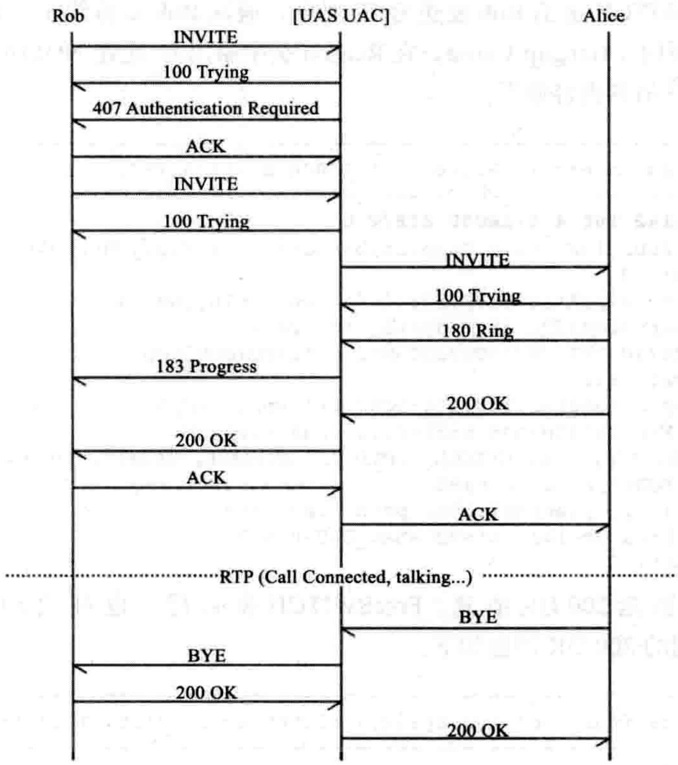


图 7-8 SIP B2BUA 接呼叫流程图

从图 7-8 可以看出，右半部分与图 7-6 是一样的，而左半部分也与图 7-6 类似。这就更好的说明了实际上有 4 个 UA（两对）参与到了通信中，4 个 UA 组成了通信中的两条腿。

7.4 深入理解 SIP

读者只要对照一下上面讲的 HTTP 协议与 SIP 协议，就会发现协议文本本身的格式很类似，不同的只是 SIP 的交互流程更丰富一些。下面我们再来跟大家一起再深学习一些与 SIP 相关的知识。

7.4.1 SIP URI

上面我们介绍了一些 FreeSWITCH 的基本概念，并通过一个真正的呼叫流程讲解了 SIP。由于实验中所有 UA 都运行在一台机器上，这可能会引起读者的不解，如果我们有三台机器，其中 192.168.1.9 是 FreeSWITCH 服务器，而 Bob 和 Alice 分别在另外两台机器上，那么情况可能是图 7-9 所示的样子。

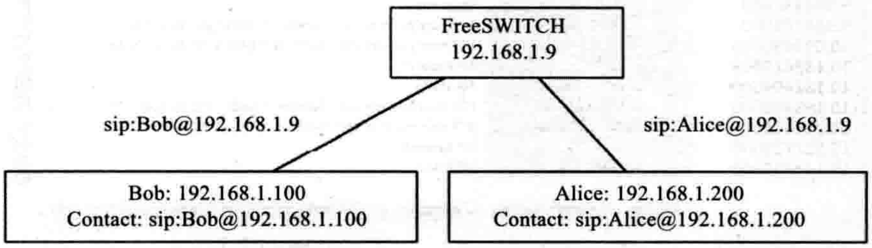


图 7-9　SIP URI

Alice 注册到 FreeSWITCH，Bob 呼叫她时，使用她的服务器地址（因为 Bob 只知道服务器地址），即 sip:Alice@192.168.1.9，FreeSWITCH 接到请求后，查找本地数据库，发现 Alice 的实际地址（Contact 地址，又叫联系地址）是 sip:Alice@192.168.1.200，便可以建立呼叫。

SIP URI 除使用 IP 地址外，也可以使用域名，如 sip:Alice@freeswitch.org.cn。更高级及更复杂的配置可能需要 DNS 的 SRV 记录，在此就不做讨论了。

这里再重复一下，Bob 呼叫 Alice 里，Bob 是主叫方，他已经知道服务器的地址，因此可以直接给服务器发送 INVITE 消息，因而它是不需要注册的[⊖]。而 Alice 不同，它是作为被叫的一方，为了让服务器能找到它，它必须事先通过 REGISTER 消息注册到服务器上。

7.4.2 SDP 和 SOA

SIP 负责建立和释放会话，一般来说，会话会包含相关的媒体，如视频和音频。媒体数

⊖ 某些服务器在收到呼叫时会检查主叫是否注册，否则不允许继续呼叫，这是策略问题，不是必须的。

据是由 SDP（Session Description Protocol，会放描述协议）描述的。SDP 一般不单独使用，它与 SIP 配合使用时会放到 SIP 协议的正文（Body）中。

会话建立时，需要媒体协商，双方才能确定对方的媒体能力以交换媒体数据。在第 8 章我们会专门讲媒体协商，在此我们通过一个简单的例子介绍一下 SDP 是如何工作的。

这里我们来看一个 FreeSWITCH 参与的单腿呼叫的例子。客户端 607 呼叫 FreeSWITCH 默认的服务 echo，它是一个回声服务，呼通后，主叫用户不仅能听到自己的声音，还能看到自己的视频（如果有的话）。在此，为了显示直观一些，我们使用了 Wireshark 抓包并进行分析，图 7-10 显示了该 SIP 呼叫的流程（关于 Wireshark 的用法我们将在第 12 章讨论）。

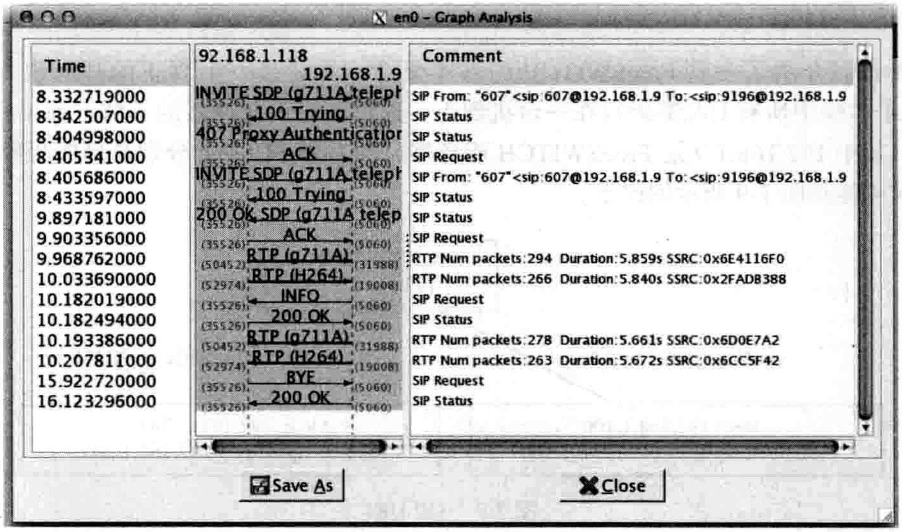


图 7-10 带 SDP 的 SIP 呼叫

图 7-10 中，客户端（192.168.1.118）呼叫 FreeSWITCH（192.168.1.9），INVITE 中带了 SDP 消息。其认证过程与我们上面讲到的类似。最后，FreeSWITCH 回复 200 OK 对通话应答，然后双方互发 RTP 媒体流（G711A，即 PCMA 的音频和 H264 的视频）。并且，在图中可以看出，客户端的 SIP 端口号是 35526，音频端口号是 50452，视频端口号是 52974；FreeSWITCH 端的端口号则分别是 5060、31988 和 19008。后面我们会在 SIP 消息中找到这些端口，这里就不再赘述了，大家只要知道用到的是这些端口就可以了。

在本章前面的例子中，为了节省篇幅，我们都省略了 SDP 部分。下面是一个完整的 SIP INVITE 消息：

```
------------------------------------------------------------------------
recv 921 bytes from udp/[192.168.1.118]:35526 at 05:48:25.379261:
------------------------------------------------------------------------
INVITE sip:9196@192.168.1.9 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.118:35526;branch=z9hG4bK-d8754z-0a09c74c6345dc09-1---d8754z-;rport
```

```
Max-Forwards: 70
Contact: <sip:607@192.168.1.118:35526>
To: <sip:9196@192.168.1.9>
From: "607"<sip:607@192.168.1.9>;tag=f49f383a
Call-ID: ZTQ0N2Y2NzI2ZjMxZTcwZTY0YTA5ODUyZDUzNWM2YjM
CSeq: 1 INVITE
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
Content-Type: application/sdp
Supported: replaces
User-Agent: Bria 3 release 3.5.0b stamp 69410
Content-Length: 381

v=0
o=- 1371880105304943 1 IN IP4 192.168.1.118
s=Bria 3 release 3.5.0b stamp 69410
c=IN IP4 192.168.1.118
b=AS:2064
t=0 0
m=audio 50452 RTP/AVP 8 0 98 101
a=rtpmap:98 ILBC/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=sendrecv
m=video 52974 RTP/AVP 123
b=TIAS:2000000
a=rtpmap:123 H264/90000
a=fmtp:123 profile-level-id=428014;packetization-mode=0
a=rtcp-fb:* nack pli
a=sendrecv
---------------------------------------------------------------------------
```

关于 SIP 头部，我们前面已经了解得差不多了。其中的 Content-Length 也跟 HTTP 协议中的类似，表示正文的长度。读者也能看出正文的类型用 Content-Type 表示，在这里它是 application/sdp，表示正文中是 SDP 消息。同样，一个空行把 SIP 头（Header）与 SIP 正文（Body）部分隔开（SIP 头部的结束是以“\r\n\r\n”为标志的）。

下面我们主要讨论 SDP 部分。

- v=：Version，表示协议的版本号。
- o=：Origin，表示源。值域中各项（以空格分隔）的涵义依次是 username（用户名）、sess-id（会话 ID）、sess-version（会话版本号）、nettype（网络类型）、addrtype（地址类型）、unicast-address（单播地址）。
- s=：Session Name，表示本 SDP 所描述的 Session 的名称。
- c=：Connecton Data，连接数据。其中值域中以空格分配的两个字段分别是网络类型和网络地址，以后的 RTP 流就会发到该地址上。注意，在 NAT 环境中如果我们要解决 RTP 穿越[⊖]问题就要看这个地址。NAT 问题将在第 9 章节中讲到。

⊖ 在 NAT 环境中，处于内部的主机又往往不知道自已在 NAT 设备中对应的公网地址，因而此处的网络地址就只好填私网的 IP 地址，而处于公网的主机无法直接访问 NAT 内部主机的 IP 地址（私网地址），因而会出现 RTP 媒体流不通的情况。

- b=：Badwidth Type，带宽类型。
- t=：Timing，起止时间。0 表示无限。
- m= ：audio Media Type，媒体类型。audio 表示音频，50452 表示音频的端口号，跟图 7-10 中的一致；RTP/AVP 是传输协议[㊀]；后面是支持的 Codec 类型，与 RTP 流中的 Payload Type[㊁]（载荷类型）相对应，在这里分别是 8、0、98 和 101，8 和 0 分别代表 PCMA 和 PCMU，它们属于静态编码，因而有一一对应的关系，而对于大于 95 的编码都属于动态编码，需要在后面使用“a=rtpmap”进行说明。
- a= ：Attributes，属性。它用于描述上面音频的属性，如本例中 98 代表 8000Hz 的 ILBC 编码，101 代表 RFC2833 DTMF 事件。a=sendrecv 表示该媒体流可用于收和发，其他的还有 sendonly（仅收）、recvonly（仅发）和 inactive（不收不发）。
- v= ：Video，视频。可以看出它的端口号 52974 也是跟图 7-10 中是一致的。而且与 H264 的视频编码对应的也是一个动态的 Payload Type[㊂]，在本例中是 123。

FreeSWITCH 收到上述的请求后，进行编码协商，这里我们省去 SIP 交互的中间环节，直接看 200（应答）消息：

```
------------------------------------------------------------------------
send 1255 bytes to udp/[192.168.1.118]:35526 at 05:48:26.939191:
------------------------------------------------------------------------
SIP/2.0 200 OK
...
Content-Type: application/sdp
Content-Length: 297

v=0
o=FreeSWITCH 1371848118 1371848119 IN IP4 192.168.1.9
s=FreeSWITCH
c=IN IP4 192.168.1.9
t=0 0
m=audio 31988 RTP/AVP 8 101
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - - -
a=ptime:20
m=video 19008 RTP/AVP 123
a=rtpmap:123 H264/90000
------------------------------------------------------------------------
```

为节省篇幅，我们省略了一部分 SIP 头。下面直接看 200 返回的 SDP 数据。在这里我

㊀ RTP/AVP 即 RTP 音频、视频 Profile，由 RFC 3551 定义，参见 http://en.wikipedia.org/wiki/RTP_audio_video_profile 及 http://tools.ietf.org/html/rfc3551 。

㊁ Payload Type 为 RTP 中承载的媒体数据的类型，简称载荷类型。RFC3551 定义了一些常用的音频、视频媒体，如 0 代表 PCMU，8 代表 PCMA 等。

㊂ 由于 Payload Type123 大于 95，因而它属于一个动态编码，即实际编码内容与该数字不是一一对应的。实际的映射关系要在后续的“a=rtpmap”行中进行说明。本例中，“a=rtpmap:123 H264/90000”表示它说明了一对 Payload Type 和实际编码的对应关系，即在这里，123 表示 H264 编码（仅在本次会话中有效）。

们也能找到音、视频的 IP 地址 192.168.1.9 以及端口号 31988 和 19008。该 SDP 也携带了 FreeSWITCH 协商后的编码 PCMA（8）以及“a=ptime”项，其中 ptime 表示 RTP 数据的打包时间，其实这里也可以省略，默认是 20 毫秒。至此，双方都有了对方的 RTP 地址和端口信息，它们就可以互发 RTP 流了。

媒体流的协商过程称为 SOA[㊀]（Service Offer and Answer，提议 / 应答）。通俗地讲，它首先由一方提供支持的 Codec 类型，由另一方选择。如本例中，607 先在 INVITE 中提议："我支持 PCMA、PCMU 和 ILBC 编码（在 m=audio 50452 RTP/AVP 8 0 98 101 行中说明），你看咱俩用哪种通信比较好？"，FreeSWITCH 在 200 OK 中回复说："那我们就用 PCMA 吧"（m=audio 31988 RTP/AVP 8 101）。具体 SOA 流程及 FreeSWITCH 选择 Codec 的策略和依据我们将在第 8 章中详细解释。

7.4.3　3PCC

值得一提的是，并不是所有的 INVITE 请求都是带 SDP 的，在 3PCC 中可能没有 SDP。

3PCC（Third Party Call Control，第三方电话呼叫控制）指的是由第三方控制者（Controller）在另外两者之间建立的一个会话，由控制者负责会话双方的媒体协商。3PCC 是一种非常灵活的会话控制方式。在 PSTN 网中，第三方呼叫控制通常用于会议、接线业务（接线员创建一个连接另外双方的呼叫）。同样，使用 SIP 协议也可以借助 3PCC 来完成 PSTN 网中的一些相关业务，例如点击拨号、通话过程中放音等等，而且实现起来非常方便。

3PCC 的实现关键就在于控制者如何使用 SOA 在会话双方之间使用 SDP 消息协商即将建立的会话。RFC3725[㊁]描述了几种实现 3PCC 的方法，我们在此以第一种为例简单介绍一下。详细了解 3PCC 已超出本书的范围，在此笔者仅以此为例帮助读者理解不带 SDP 的 INVITE 呼叫的交互流程，而对 3PCC 则避重就轻。一个典型的 3PCC 呼叫如图 7-11 所示。

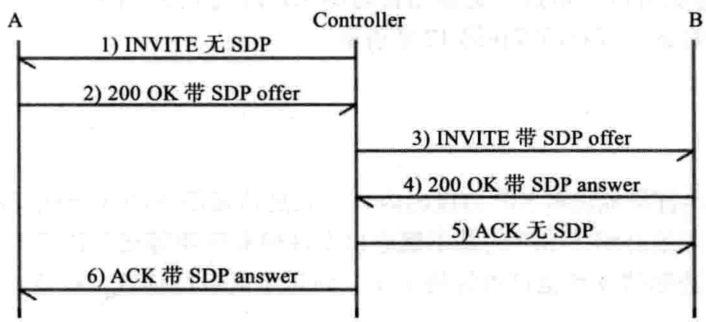


图 7-11　一个简单的 3PCC 呼叫

其中：

1）Controller 向 A 发送一个不带 SDP 的 INVITE 请求。

㊀ 它是在 RFC3264 中定义的，参见 http://tools.ietf.org/html/rfc3264。

㊁ http://tools.ietf.org/html/rfc3725。

2）A 给 Controller 回送 200 OK，其中的 offer 指的是 SDP 消息，其中还有描述与 A 相关的媒体处理能力的数据，详见 SDP 协议。

3）Controller 向 B 发送 INVITE 请求，这其中带有 A 的 SDP 相关描述。

4）B 给 Controller 回送 200 OK 应答，其中的 answer 指的是 SDP 消息，其中描述与 B 相关的媒体处理能力的数据。

5）Controller 给 B 回送 ACK，不含任何 SDP 信息，因为在前面的 INVITE 中已经有了。

6）Controller 对 A 回送 ACK，其中带有 B 的 SDP 相关信息，这其实就可使媒体流在 A 和 B 之间传输了，而不用 Controller 去转发媒体流。

这就是用一个第三方的 Controller 进行呼叫控制的例子。FreeSWITCH 默认不支持没有 SDP 的 INVITE 请求，如果需要响应这种请求，则可以尝试在 SIP Profile 中开启 enable-3pcc 参数。

7.4.4 SIP 承载

大家已经熟知，HTTP 是用 TCP 承载的，而 SIP 则支持 TCP 和 UDP 承载。事实上，RFC3261 规定，任何 SIP UA 必须同时支持 TCP 和 UDP[一]。我们常见的 SIP 都是用 UDP 承载的，由于 UDP 是面向无连接的，在大并发量的情况下与 TCP 相比可以节省 TCP 由于每个 IP 包都需要确认带来的额外开销[二]。不过，在 SIP 包比较大的情况下，如果超出了 IP 层窗口的大小（通常是 1500），在经过路由器的时候可能会被拆包，使用 UDP 承载的 SIP 消息就可能发生丢失、乱序等，这时候就应该使用 TCP。

在需要对 SIP 加密的情况下，可以使用 TLS[三]。TLS 是基于 TCP 的。

在新的网络时代，又出了一个新的草案，称为 SIP over WebSocket[四]。当前有些浏览器如 Chrome 和 FireFox 已经实现了 WebSocket，从而可以通过它承载 SIP；而它们也实现了 WebRTC[五]，这意味着可以通过浏览器与普通的 SIP 话机（甚至 PSTN）进行音频、视频通话。关于 WebRTC 的内容我们将在第 12 章讲解。

7.5 小结

本章没有照本宣科地讲解 SIP 协议的内容，而是从最简单的注册和呼叫开始，以讲故事的方式由浅入深地介绍了 SIP 的基本概念以及注册和呼叫等交互流程。希望没有 SIP 甚至也没有 HTTP 基础的读者也可以轻松上手。而对于高级的读者，本章也给出了延伸阅读

[一] All SIP elements MUST implement UDP and TCP. SIP elements MAY implement other protocols.

[二] 当然，仁者见仁，智者见智。笔者也听有人说过由于使用了 TCP，客户端可以不用总发 REGISTER 注册消息以用于保持连接或穿越 NAT 等，反而节省了资源。可惜找不到原文出处了。

[三] http://zh.wikipedia.org/wiki/ 安全套接层。

[四] http://datatracker.ietf.org/doc/draft-ietf-sipcore-sip-websocket/。

[五] http://www.webrtc.org/ http://zh.wikipedia.org/wiki/WebRTC。

的资料。

FreeSWITCH 是一个 B2BUA。读者在了解了 FreeSWITCH 的架构以后，再结合 SIP 的详细呼叫流程思考一下，就比较容易理解了。

本章的协议文本都是从 FreeSWITCH 中实际抓出来的，看起来比较直观。读者可以在学习的过程中自行练习。获取这些 SIP 消息的方法是在 FreeSWITCH 命令行上执行如下命令：

```
freeswitch> sofia global siptrace on
```

然后，打电话看一下能不能在控制台上看到信令。有时候，注册的电话多了以后，信令消息比较多，看起来就比较累。在这种情况下笔者一般是执行 fs_cli 连接到 FreeSWITCH，打开 siptrace，打一个测试电话后就立即用 Ctrl+D（或 /exit）退出 fs_cli，然后慢慢分析。当然对初学者而言还是建议找一个清净的环境学习，不要注册太多分机。

如果需要关闭 trace，可执行如下命令：

```
freeswitch> sofia global siptrace off
```

最权威的资料还是 RFC 3261，如果你从事这方面的开发，还是需要好好读一读的。当然，如果你觉得 RFC 比较枯燥的话，网上也有一些有趣的关于 SIP 的阅读资料，感兴趣的读者不妨去读一读。这些资料有：

- http://www.freeswitch.org.cn/2012/06/20/guan-yu-voipni-ceng-jing-yi-wei-ni-bu-bi-zhi-dao-de-suo-you-dong-dong.html
- http://arstechnica.com/business/2010/01/voip-in-depth-an-introduction-to-the-sip-protocol-part-1/
- http://arstechnica.com/tech-policy/2010/03/voip-in-depth-an-introduction-to-the-sip-protocol-part-2/

第 8 章

媒　体

通过第 7 章，我们已经了解了 SIP 信令，它用于帮助我们建立会话。但我们建立通话的目的是"通话"，即通话的双方要能互相听到对方的声音。通过第 3 章我们也已经了解到，要通过采样、量化、编码将我们声音的模块量变成数字信号，以便在数字线路上传输，我们把这些数字信号称为媒体（Media）。

本章我们就一起来学习一下媒体的知识，以及 FreeSWITCH 中的媒体处理机制和协商策略。

8.1　媒体与媒体处理

常见的媒体有音频、视频、图像（传真）、文本等。为了简单起见，接下来我们主要以音频为例来讲解相关的知识以及在 FreeSWITCH 中的实现和配置等。

8.1.1　音频编码

从模拟信号变成数字信号的过程称为模数转换（Analog Digital Convert，AD）。AD 转换要经过采样、量化、编码三个过程。编码（Code）就是指按照一定的规则将采样所得的信号用一组二进制或者其他进制的数来表示。经过编码后的数据便于在数字网络上传输，到达对端以后，再通过解码（Decode）过程变成原始信号，进而经过数模转换（Digital Analog Convert，DA）再恢复为模拟量，即转换为人们能够感知的信号。

一般来说，编码与解码过程都是成对出现的，所以习惯上人们把它们合起来说，称为编解码（Codec），即 Co（de）与 Dec（ode）的合成写法，但有时候为了方便，也简称为编码，如我们常说的音频编码或视频编码。

1. PCM 编码

我们在 1.2.2 节讲过 PCM。使用 PCM 方式对原始声音信号进行采样、量化后得到线性编码，然后再进行压缩，这种编码方式就称为 PCM 编码。PCM 的两种压缩方式 A 律和

μ 律（alaw 和 ulaw）对应的编解码名称分别为 PCMA 和 PCMU ㊀。PCM 编码是在 ITU-T Recommendation G.711 标准中定义的，因而又称为 G.711 ㊁编码。

经过压缩的 PCM 编码仍然占用比较高的带宽，因此在带宽比较紧张或比较昂贵的场合（如卫星线路上）又出现了一些更高级的压缩算法，目的都是降低带宽、提高传输效率。

另外，由于 VoIP 是在 IP 网上传输的，而大多数的网络没有 QoS（Quality of Service）保证，容易产生丢包（Packet Loss）、延迟（Delay）或抖动（Jitter）。它们对于普通数据传输影响不是很严重，但对于实时的语音来说，影响就比较大，有时会引起声音质量的迅速下降，严重时甚至无法分辨。因此，某些好的压缩算法又平衡了这些因素带来的影响，在质量不好的网络上也能得到较好的声音质量。

PCM 编码的采样频率为 8000Hz，而随着技术的进步及人们对声音质量要求的提高，各种高清（High Definition，HD）编码也纷纷涌现，如 G.722 等。

如果在网络上传输语音，则要将编码后的语音数据打包。通常使用的打包时间间隔为 20ms，即将 20ms 的音频数据放到一个数据包里传送，也可以理解为每 20ms 打一个包。如果采样频率是 8000Hz，那么 1 秒就能传输 1000ms/20ms=50 个包，每个包携带 8000/50=160 个采样数据。在 PCMA 或 PCMU 方式中，每个采样数据占 1 字节，因此一个 20ms 的 PCM 包的数据净荷就是 160 字节。

2. FreeSWITCH 支持的其他语音编码

FreeSWITCH 支持非常丰富的语音编码，具体如表 8-1 所示。

表 8-1 FreeSWITCH 支持的语音编码

编 码	说 明
G.711	即 PCM，基本的语音编码方式，分为 PCMU 和 PCMA 两种
G.722	支持高清（HD）语音
G.726	使用差分脉冲编码调制（ADPCM）
G.723.1	低带宽编码 5.3 ~ 6.3kbit/s
G.729	低带宽 8kbit/s，有 G729a 和 G729b 两种㊂
GSM	13.3kbit/s，通常认为语音质量比 G.729 差一个数量级
iLBC	13.3kbit/s、15.2kbit/s，来自 GIPS ㊃，抗丢包

㊀ 这些名字用于在 SDP 中标志 RTP 中传输的媒体的类型，参见：http://en.wikipedia.org/wiki/RTP_audio_video_profile。

㊁ 参见 http://en.wikipedia.org/wiki/G.711 。

㊂ 与其他压缩算法不同的是，G.729 携带的不是真正语音数据的压缩结果，而是使用 CS-ACELP 预测模型得到的声音码表。CS-ACELP 的全称是 Conjugate-Structure Algebraic-Codec-Excited Liner，它利用数学模型模拟人类的各种发声方式，可以建立声音码表，用发送对应的声音代码代替实际的声音采样。对端收到后再根据这些声音代码合成语音。

㊃ Global IP Sound，后改为 Global IP Solutions，在语音编解码方面颇有建树，它们的解决方案在抗丢包及回声消除方面做得特别好，广泛应用于我们所熟知的产品，如 Skype 及 QQ。2010 年被 Google 收购，后来开源了 iSAC。

（续）

编 码	说 明
iSAC	来自 GIPS，为宽带（16kHz）编码
CODEC2	2550bit/s，最省带宽的
Speex	支持窄带（8kHz）和宽带（16kHz）编码，Flash Player 支持该编码
SILK	来自 Skype，抗丢包
CELT	低延迟高清编码，48kbit/s，能传输采样率为 48kHz 的语音，达到 CD 音质
OPUS	综合 CELT 和 SILK 之长，达到宽、窄带和高、低带宽的绝佳组合

在 FreeSWITCH 中，编码名称的格式为“名称 @xxh@yyi”。其中 h 代表 Hz，即采样频率；i 则代表 ptime，即打包间隔；xx 和 yy 分别代表实际的数值，如 speex@16000h@20i 表示 16000Hz、20ms 的 Speex 编码。表 8-2 列举了 FreeSWITCH 中各种编码的一些详细参数。

表 8-2 FreeSWITCH 支持的语音参数

名 称	速率 /(kbit/s)	频率 /kHz	打包时间 /ms	说 明
PCMA	64	8	10~120 ⊖	A 律（aLaw）
PCMU	64	8	10~120	μ 律（μ Law）
iSAC	10,32	16,32	30,60	
CODEC2	2.55	8	20	
SILK	20,25,30,40	8,12,16	20	Skype 语音编码
iLBC	13.3,15	8	20,30	iLBC 30ms 的语音在绝大多数情况下都是最佳选择
Speex	24.6,42.2,44	8,16,32	20	
BV32	32	16	20	BroadVoice 宽带
BV16	16	8	20	BroadVoice 窄带
G722	64	16	10~60	
G7221@16000h	24,32	16	20,40,,60	Siren 7
G7221@32000h	48	32	20	Siren 14
CELT@32000h	48	32	2,4,6,8,10	
CELT@48000h	48	48	2,4,6,8,10	
DVI4	32,64	8,16	10~120	
GSM@40i	8	8	20,40,60	
G726-16	16	8	10~120	
G726-24	8	16	10~120	
G726-32	8	32	10~120	
G726-40	40	8	10~120	
AAL2-G726-16	16	8	10~120	使用 AAL2 打包方案

⊖ 以 10ms 为间隔，即 10,20,30,···,120，下同。

（续）

名　称	速率 /(kbit/s)	频率 /kHz	打包时间 /ms	说　明
AAL2-G726-24	24	8	10~120	使用 AAL2 打包方案
AAL2-G726-32	32	8	10~120	使用 AAL2 打包方案
AAL2-G726-40	40	8	10~120	使用 AAL2 打包方案
LPC	2.4	8	90	LPC10
OPUS	10,20,30,40	8,48	10,20,30,40	OPUS
B64	20	8,16,32	20	Base64 编码
L16	128,256,512,768	8,16,32,48	10~120	L16 在 VoIP 中不推荐使用，虽然也可以用，但它的大小可能会超过网络的 MTU

表 8-3 中所列的编码，由于专利原因，只能透传。其中，G729 编码在 FreeSWITCH 中 Linux 平台上有 Licence 解决方案，购买一路需要 10 美元，其中包含一个编码器和一个解码器。

表 8-3　透传的语音编码

名称	速率 /(kbit/s)	频率 /kHz	打包时间 /ms	说明
G729	8	8	10 ~120	透传
G723	6.3	8	30,60,90,120	透传
AMR	12.2	8	20	透传
AMR-WB				G.722.2

另外，关于这些编码的比较，可以参见 http://en.wikipedia.org/wiki/Comparison_of_audio_codecs。

3. FreeSWITCH 中与编码相关的主要命令

如果要查看 FreeSWITCH 支持的编码类型，可以在控制台上使用 show codec 命令，下面是该命令在笔者电脑上的输出结果（有删节）。

```
freeswitch> show codec

codec,ADPCM (IMA),mod_spandsp
codec,AMR,mod_amr
codec,G.711 alaw,CORE_PCM_MODULE
codec,G.711 ulaw,CORE_PCM_MODULE
codec,G.722,mod_spandsp
codec,G.723.1 6.3k,mod_g723_1
codec,G.726 16k,mod_spandsp
codec,G.729,mod_g729
codec,H.264 Video (passthru),mod_h26x
codec,VP8 Video (passthru),mod_vp8
```

上述结果中，每一行代表某一个编码，都标明了相关的编码名称及参数（以逗号隔开的第 2 列）以及具体实现的模块（第 3 列）。我们可以看到，G.711 编码（A 律和 μ 律）都是在核心模块（CORE_PCM_MODULE）中实现的。关于具体的实现代码，我们在第 18 章讲

源代码阅读的时候还会讲到。

另外，在 FreeSWITCH 启动的时候，或者你使用 reload 来重新加载某个编解码模块的时候，可以看到控制台上有类似如下的输出（下面是重新加载 mod_spandsp 时的输出，该模块提供了很多语音编码，有删节）。

```
[NOTICE] Adding Codec DVI4 6 ADPCM (IMA) 16000hz 40ms 64000bps
[NOTICE] Adding Codec DVI4 6 ADPCM (IMA) 16000hz 50ms 64000bps
[NOTICE] Adding Codec DVI4 6 ADPCM (IMA) 16000hz 60ms 64000bps
[NOTICE] Adding Codec DVI4 5 ADPCM (IMA) 8000hz 10ms 32000bps
[NOTICE] Adding Codec DVI4 5 ADPCM (IMA) 8000hz 20ms 32000bps
[NOTICE] Adding Codec DVI4 5 ADPCM (IMA) 8000hz 30ms 32000bps
[NOTICE] Adding Codec G726-16 124 G.726 16k 8000hz 10ms 16000bps
[NOTICE] Adding Codec G726-16 124 G.726 16k 8000hz 20ms 16000bps
[NOTICE] Adding Codec G726-16 124 G.726 16k 8000hz 30ms 16000bps
....
```

上述结果中，上面每一行都代表增加了一种编码。它们的意义一目了然，因此，在这里就不多解释了。

在实践中，当你不清楚某种编码所提供的各种参数时，可以尝试重新加载所属模块。比如 SILK 编码属于 mod_silk 模块，则可以使用 reload mod_silk 实现，示例如下：

```
freeswitch> reload mod_silk

[NOTICE] Adding Codec SILK 119 SILK 16000hz 20ms 30000bps
[NOTICE] Adding Codec SILK 118 SILK 12000hz 20ms 25000bps
[NOTICE] Adding Codec SILK 117 SILK 8000hz 20ms  20000bps
```

iLBC 编码也是由独立的模块提供的，重新加载的命令和输出如下：

```
freeswitch> reload mod_ilbc

[NOTICE] Adding Codec iLBC 97 iLBC 8000hz 30ms 13330bps
[NOTICE] Adding Codec iLBC 98 iLBC 8000hz 20ms 15200bps
```

音频编码最基本的两个技术参数就是采样频率和打包周期。采样频率越高，声音就越清晰，保留的细节就越多，当然它会占用更大的带宽。对于普通“人声”通话来讲，8000Hz 就够了，但对于高品质的音乐来讲，就需要更高的采样率才能保证“悦耳”，比如我们通常说的 CD 音质的声音使用的就是 44.1kHz 的采样率。打包周期跟传输有关，打包周期超短，延迟越小，相对而言传输开销就会越多，因而需要更大的带宽；打包周期超长，带来的延迟就越大，如果传输过程中有丢包，对语音质量的影响也就越大。大部分编码都支持多种打包周期，最常见的是 20ms，iLBC、G.723 等默认使用 30ms，更长的打包周期如 60~120ms，通常用于卫星链路等高延迟、低带宽的场合。

注意，在 FreeSWITCH 中，有些编码不是默认安装的。如果要使用这些编码，需要自己编译，如在源代码目录下可执行以下命令来安装相应的编码（中文注释为笔者所加）：

```
# make mod_celt-install          # 安装 CELT 编码
# make mod_silk-install          # 安装 SILK 编码
# make mod_isac-install          # 安装 iSAC 编码
```

```
# make mod_opus-install          # 安装 OPUS 编码
# make mod_codec2-install        # 安装 CODEC2 编码
```

8.1.2　媒体工作机理和相关配置

为了帮助大家更好地理解媒体，下面我们简单介绍一下媒体的工作机理和相关的配置。

1. 工作机理

在第 7 章中，我们只讨论了 SIP 的呼叫建立和释放流程，而没有考虑媒体的情况。实际上，在基于 SIP 的通信中，媒体数据是在 RTP 流中传输的。

以 PCM 音频数据为例，我们上面已经讲过，如果打包周期是 20ms，则每个音频包得到 160 字节的数据，在这些数据前面加上 12 字节的 RTP 包头，就成为了一个 RTP 包。RTP 包头中携带了音频的编码类型及时间戳等数据，以便于对方收到后进行解码回放及同步等。

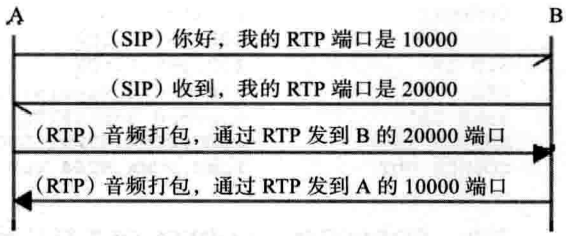


图 8-1　简化的 SIP/RTP 交互流程示意图

RTP 包使用与 SIP 不同的 UDP 端口传送，因而在实际传输前需要先通过 SIP 信令与对方“协商”好该往哪个端口传送，一个高度简化的 SIP/RTP 交互流程示意图如图 8-1 所示。从图中我们可以看到，A 和 B 首先通过 SIP 信令“协商”好双方将要使用的 RTP 端口号（当然还有双方的 IP 地址，我们省略了），然后双方互相向对方发送 RTP 媒体流。

SIP 可使用 TCP 或 UDP 承载，但 RTP 数据一般只使用 UDP 承载。更详细的协商过程我们稍后讨论，下面先来看一下 FreeSWITCH 中与媒体相关的配置。

2. 相关配置

在默认的配置中，SIP Profile 支持的媒体列表是在 vars.xml 文件中配置的，如：

```
<X-PRE-PROCESS cmd="set" data="global_codec_prefs=G722,PCMU,PCMA,GSM"/>
<X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=PCMU,PCMA,GSM"/>
```

如果要增加其他编码的支持（如 G729），可以将上述配置项改为：

```
<X-PRE-PROCESS cmd="set" data="global_codec_prefs=G722,PCMU,PCMA,GSM,G729"/>
<X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=PCMU,PCMA,GSM,G729"/>
```

然后需要重启 FreeSWITCH 以使设置生效⊖。

如果在学习过程中需要频繁做实验，不妨直接修改 Profile 配置中的相关项，如在 internal.xml 中，将下面的行：

```
<param name="inbound-codec-prefs" value="$${global_codec_prefs}"/>
```

⊖ 在 UNIX 类平台上，也可以给 FreeSWITCH 发送 HUP 信号让 FreeSWITCH 重新解析这些 XML 中的全局变量。首先找到 FreeSWITCH 的 Pid，然后 kill -HUP <Pid>，最后在 FreeSWITCH 中执行 sofia profile internal rescan。

改为：

```
<param name="inbound-codec-prefs" value="PCMU,PCMA,G729,iLBC,H264,VP8"/>
```

然后只需在 FreeSWITCH 控制台上或 fs_cli 中执行如下命令使刚才的配置生效：

```
freeswitch> sofia profile internal rescan
```

以下命令可以列出当前与 Profile 相关的配置情况，如（有删节）：

```
freeswitch> sofia status profile internal

=================================================================
Name                internal
Dialplan            XML
Context             public
RTP-IP              192.168.1.102
SIP-IP              192.168.1.102
URL                 sip:mod_sofia@192.168.1.102:5060
BIND-URL            sip:mod_sofia@192.168.1.102:5060
CODECS IN           PCMU,PCMA,G729,iLBC,H264,VP8
CODECS OUT          PCMU,PCMA,H264,VP8
……
```

其中，CODECS IN、CODECS OUT 行显示了呼入、呼出的编码设置。

8.2 媒体协商

世界是丰富多彩的，不同的 SIP 终端有不同的特性。支持不同的语音编码。所以不同的 SIP 终端进行通信时需要先与支持的编码进行“协商”，以便双方互相能够“理解”对方发来的媒体流中的数据。

8.2.1 协商过程

我们来看一个最简单的编码协商过程。如果在 FreeSWITCH 中，按 F8 打开详细的 Log，打一个电话，就会在 Log 中看到如下类似的协商过程：

```
Audio Codec Compare [iLBC:97:8000:60:0]/[G722:9:8000:20:64000]
Audio Codec Compare [iLBC:97:8000:60:0]/[G729:18:8000:20:8000]
Audio Codec Compare [iLBC:97:8000:60:0]/[PCMU:0:8000:20:64000]
Audio Codec Compare [iLBC:97:8000:60:0]/[PCMA:8:8000:20:64000]
Audio  Codec Compare  [iLBC:97:8000:60:0]/[GSM:3:8000:20:13200]
```

在上面的例子中，SIP 客户端呼入时使用 iLBC 编码，打包周期是 60ms，Payload Type 是 97；而服务器端提供 G722、PCMU、PCMA、GSM 等选择，但都是 20ms 的。所以我们在上述 Log 中可以看出 FreeSWITCH 会将客户端提供的编码与本地提供的编码逐一比较。由于客户端与服务器提供的编码没有交集，故上面的协商是不成功的，在这种情况下，FreeSWITCH 会返回 SIP 488 消息，失败原因 INCOMPATIBLE_DESTINATION（目的地不兼容）说明找不到兼容的编码。

SIP 采用 Offer/Answer（请求 / 应答）机制来协商。请求发起的一方提供（Offer）自己支持的媒体编码列表，被请求的一方比较自己支持的媒体列表最终选择一种（或几种）编码以应答（Answer）方式通知请求者，然后它们就可以使用兼容的编码进行通信了。下面是一次成功的协商 Log（客户端使用 PCMA，最终成功协商 PCMA）：

```
Audio Codec Compare [PCMA:8:8000:20:64000]/[G722:9:8000:20:64000]
Audio Codec Compare [PCMA:8:8000:20:64000]/[PCMU:0:8000:20:64000]
Audio Codec Compare [PCMA:8:8000:20:64000]/[PCMA:8:8000:20:64000]
switch_core_media.c:3213 Audio Codec Compare [PCMA:8:8000:20:64000]
++++ is saved as a match
Set Codec sofia/internal/1006@192.168.7.6 PCMA/8000 20 ms 160 samples 64000
```

从上面的协商 Log 中可以看到，它也是将客户端的编码与服务器端的编码进行逐一比较，最后一行的“Set Codec … PCMA”表示本次协商成功并把该 Channel 的编码设为 PCMA 编码。

8.2.2 SDP 及其在编码协商中的作用

为了更直观地理解 SIP 协议中的编码协商过程，下面我们来详细看一下相关的 SIP 信令。

由第 7 章我们知道，SIP 信令格式类似 HTTP 协议，由请求 / 应答行、消息头及消息体构成。一般的 SIP INVITE 消息体都包含 SDP[⊖]，我们来看一个 INVITE 请求（为节省篇幅，消息头部分做了删节），消息如下：

```
INVITE sip:3000@192.168.1.102 SIP/2.0
To: <sip:3000@192.168.1.102>
From: "1000"<sip:1000@192.168.1.102>;tag=53604628
Content-Type: application/sdp
Content-Length: 222

v=0
o=- 1364745810658806 1 IN IP4 192.168.1.102
s=Bria 3 release 3.5.0b stamp 69410
c=IN IP4 192.168.1.102
b=AS:2064
t=0 0
m=audio 59820 RTP/AVP 8 18 101
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=sendrecv
```

其中，媒体类型是由 SDP 消息（Session Description Protocol，会话描述协议）描述的，SDP 消息的 MIME 类型是 application/sdp，它是在 SIP 头中以 Content-Type 标志，并在 SIP 消息的正文（Body，又称消息体，消息头中的 Content-Length 指示 Body 的长度）中携带。以下我们将忽略 SIP 消息头部分，只讨论 SDP 消息。

SDP 消息从 v=0 一行开始。“C=IN”一行表示媒体所在机器的 IP 地址。“m=audio”

⊖ 如在 3PCC 应用中，有些 INVITE 消息中可能没有 SDP。参见 7.4.3 节。

一行描述音频数据，其中 59820 就是 RTP 的端口号，该端口用于收、发 RTP 数据；RTP/AVP 表示使用 RTP 承载的 Audio/Video Profile[⊖]；8 和 101 表示音频编码的代号，它是由 IANA[⊜]负责分配和管理的，因而又称为 IANA 代码，在此 8 代表 PCMA，18 代表 G729。a=sendrecv 表示双向收发，其他的还有 sendonly（只发）、recvonly（只收）及 inactive（不发不收）等。

当 FreeSWITCH 收到后，即启动协商过程。如上一节的 Log 所示，服务器端提供 PCMU 和 PCMA，因此当比较到 PCMA 时协商成功，FreeSWITCH 将返回如下 SIP 消息：

```
SIP/2.0 200 OK
... (略)
Content-Type: application/sdp
Content-Length: 249

v=0
o=FreeSWITCH 1364717298 1364717299 IN IP4 192.168.1.102
s=FreeSWITCH
c=IN IP4 192.168.1.102
t=0 0
m=audio 28512 RTP/AVP 8 101
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - - -
a=ptime:20
```

以上消息中的 SDP 说明 FreeSWITCH 这端的 RTP 端口号是 28512，使用 PCMA 编码。至此，协商完毕，双方互相知道了对方的 IP 地址和端口号，就可以互发音频 RTP 包了。

注意，在本例中，FreeSWITCH 回复的是 200 消息以此作为对 Offer 的 Answer，当然具体回复的消息要视具体情况而定，如果在早期媒体的情况下则需要先回复，这时可能是 183 消息。我们第 7 章已经讲过，1xx 消息是呼叫进展消息。当 UA 发出一个请求后，如果对方回复的是 180，证明呼叫进展顺利，主叫这一端即自己产生回铃音给用户一个有效的反馈。如果对方回复的是 183，则 183 是可以携带 SDP 的，从而可以在协商成功后把自己个性化的媒体也传送过来，如彩铃或呼叫失败之前播放的录音通知等，这些媒体称为早期媒体（即 Early Media，与正常的通话数据相比，它来得早了那么一点点）。对方在回送 180 或 183 时，自己的话机开始振铃，如果有人接听，则切断回铃音，并发送 200 OK 消息，一般来说交换机会在此时开始计费，因为正常的通话开始了。

另外，上面 SDP 消息中有一行“a=rtpmap:8 PCMA/8000”，它是对其上一行中音频代码列表中“8”的描述，在本例中，这一行不是必需的。但是 IANA 代码只规定了 96 以下的代码的一一对应关系，大于或等于 96 的称为动态编码，动态编码不能只用代码来区分，还必须同时使用其名称，因而，对于动态编码来讲，rtpmap 行就是必需的。而 UA 在进行编码协商时，对于大于或等于 96 的编码，必须使用其名称来对比（如 iLBC）。也许更糟糕的是，

⊖ 参见 http://en.wikipedia.org/wiki/RTP_audio_video_profile。

⊜ the Internet Assigned Numbers Authority，互联网号码分配局。

若双方约定了 iLBC，一方的代码是 96，另一方的代码是 98，那么在 RTP 中，它们就必须使用自己的代码来发送，却需要以对方指定的代码来接收。当然，虽然代码不同，但双方都知道这是 iLBC，所以编码和解码都没有问题。很遗憾的是，有一些 UA 没有充分理解这一点，因而在互联互通时可能会出问题。

8.2.3 协商时机与策略

FreeSWITCH 的协商可分为早协商和晚协商。当呼叫到达一个 SIP Profile 时，即某端口收到 INVITE 请求而未到达路由阶段时，就先行协商的，称为早协商。而如果等到路由阶段，到达拨号计划时再进行协商的，称为晚协商（Later Negotiation）。通过使用晚协商，用户可以有更多的自由去确定协商策略。

系统默认为晚协商，如果要使用早协商，可以在 SIP Profile 中设置以下参数[⊖]：

```
<param name="inbound-late-negotiation" value="false"/>
```

这里需要注意的是，如果启用了 inbound-zrtp-passthru（默认），则隐含设置 inbound-late-negotiation 为 true。

FreeSWITCH 支持如下协商策略：generous、greedy 和 scrooge。

我们先看 generous（普通的），它是默认的协商策略。假设客户端提供的编码列表为 PCMA、G729，而 FreeSWITCH 支持的列表优先顺序为 G729、PCMU、PCMA。那么当 FreeSWITCH 收到请求时将优先考虑客户端的感受，因而优先使用 PCMA 编码。

如果协商策略设为 greedy（贪婪的），则上面的例子中 FreeSWITCH 会优先考虑自己的编码，因而会选择 G729。

如果设置成 scrooge（吝啬的），则 FreeSWITCH 会更加强势，它除了具有 greedy 所有的特性外，还将强制使用自己的采样率。

上面提到的是 UA 跟 FreeSWITCH 进行单腿协商的情况。我们总是提到 FreeSWITCH 是一个 B2BUA。在预计通话会有两条腿的情况下，启用晚协商会得到更好的效果。比如，在 SIP Profile 中设置 disable-transcoding 为 true 时，FreeSWITCH 将首先呼叫 bleg，并仅 Offer aleg 提供的编码，从而可以避免转码，以节约 CPU 资源。

8.3 其他媒体相关的问题

FreeSWITCH 有很强的媒体处理能力，也支持非常灵活的配置。下面我们再讲几个与媒体有关的概念及相关知识。

8.3.1 RTP 和 RTCP

在完成 SIP 及 SDP 协商后，真正的语音数据是在 RTP 协议中传送的。即双方在前面的

⊖ 我们将在下一章深入讲解 mod_sofia 中 Profile 中的配置。

SDP 协商中都已经获得了对方的 IP 地址、端口号，以及支持的媒体类型。接下来，就是将本地的媒体数据以指定的编码格式进行编码并通过 RTP 协议发送到对方去。

RTP ㊀协议的全称是 Real-time Transport Protocol，即实时传输协议，它是由 IETF 的多媒体传输工作小组在 RFC 3550 ㊁中定义的。RTP 协议详细说明了在互联网上传递音频和视频的标准数据包格式。它一开始被设计为一个多播协议，但后来被用在很多单播应用中。RTP 协议是建立在 UDP 协议之上的，常用于流媒体系统、视频会议和一键通（Push to Talk）系统（配合 H.323 或 SIP），它已成为 IP 电话产业的技术基础。

RFC 3550 中规定的 RTP 封包格式如下：

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

一般来说，在没有任何扩展的情况下，RTP 包头的长度为 12 字节。其中主要字段的含义简介如下：

- version (V)：1 bit。标明 RTP 版本号。RFC3550 中规定的版本号为 2。
- padding (P)：1 bit。如果该位被设置，则将在该 packet 末尾包含额外的附加信息，附加信息的最后一个字节表示额外附加信息的长度（包含该字节本身）。该字段之所以存在是因为一些加密机制需要固定长度的数据块，或者为了在一个底层协议数据单元中传输多个 RTP packets。
- extension (X)：1 bit。如果该位为 1，则在固定的头部后存在一个扩展头部，格式定义在 RFC3550 中。
- CSRC count (CC)：4 bit。在固定头部后存在多少个 CSRC 标记。
- marker (M): 1 bit。该位的功能依赖于 RTP 中实际传输媒体类型的 Profile 的定义。Profile 可以改变该位的长度，但是要保持 marker 和 payload type 总长度不变（一共是 8 bit）。一般来说，音频跟视频中该位的定义是不同的。在视频中，marker 值为 1 通常表示一帧图像的结束㊂。
- payload type (PT)：7 bit。标记着 RTP packet 所携带信息的类型，标准类型列在 RFC3551 中。如果接收方不能识别该类型，必须忽略该包。

㊀ 参见 http://zh.wikipedia.org/wiki/ 实时传输协议。

㊁ 参见 http://www.ietf.org/rfc/rfc3550 。

㊂ 如传输 H264 视频的情况，由于视频数据量较大，一帧视频数据可能会分到不同的 RTP 包中传送，同一帧数据的时间戳是相同的，若该位为 1 则该帧结束，下一个数据包的时间戳将会改变。

- sequence number：16 bit。序列号，每个 RTP 包发送后该序列号加 1，接收方可以根据该序列号检测丢包或重新排列数据包顺序等。
- timestamp：32 bit。时间戳。反映 RTP 包中所携带信息包中第一个字节的采样时间。
- SSRC：32 bit。数据源标识。在一个 RTP Session 期间每个数据流都应该有一个不同的 SSRC。如果视频源发生变化，则应该改变 SSRC，以让接收端感知到该变化。
- CSRC list：0 到 15 项，每个源标识为 32 bit。贡献数据源标识。只有存在多路混发的时候才有效。多路混发的情况，如一个将多声道的语音流合并到一个 RTP 流中发送，在这里就可以列出原来每个声道的 SSRC。

实时传输控制协议[㊀]（Real-time Transport Control Protocol 或 RTP Control Protocol，RTCP）是实时传输协议（RTP）的一个姊妹协议，前者由 RFC 3551[㊁]定义。RTP 使用一个偶数 UDP 端口，而 RTCP 则使用 RTP 的下一个相邻的奇数端口。RTCP 除为 RTP 媒体流提供信道外（out-of-band）的控制。RTCP 本身并不传输数据，但会和 RTP 一起协作将多媒体数据打包并发送出去。RTCP 定期在多媒体会话参加者之间传输控制数据。RTCP 的主要功能是为 RTP 提供的服务的质量（Quality of Service）提供反馈信息。RTCP 收集相关媒体连接的统计信息，例如传输字节数、传输分组数、丢失分组数及拉动（jitter）、单向和双向网络延迟等等，网络应用程序即可利用 RTCP 的统计信息来控制传输的品质，比如当网络拥塞比较严重时，可以限制信息流量或改用压缩率较高的编解码器。

8.3.2 转码

FreeSWITCH 是一个 B2BUA，因而在桥接两条腿时，如果两条腿分别使用不同的编码，则需要经过一个转码过程分别转成对方需要的编码。

当需要转码时，FreeSWITCH 会将收到的音频数据转成一种中间格式，称为 L16，即线性 16 位的编码。这种格式可以与其他各种编码进行转换。

另外，即使呼叫的双方采用同样的编码，但如果有 IVR 或录、放音等中间环节时，也需要转码。

有一些编码，如 G729，由于专利原因，不能自由使用，因此 FreeSWITCH 默认不支持对它的转码。如果确实需要转码，则可以购买商业版的转码模块。

FreeSWITCH 默认的配置（本例中的配置基于 FreeSWITCH 1.2.11+git~20130720）不支持自动转码，笔者用默认的配置做了一个实验，A 呼叫 B，A 端只有 PCMA，B 端只支持 PCMU，而 FreeSWITCH 使用默认的编码，即“G722,PCMU,PCMA,GSM”。A 端呼叫到达 FreeSWITCH 后，FreeSWITCH 又向 B 发送 INVITE，其中的 SDP 如下：

```
v=0
o=FreeSWITCH 1374385260 1374385261 IN IP4 192.168.7.8
s=FreeSWITCH
c=IN IP4 192.168.7.8
```

㊀ 参见 http://zh.wikipedia.org/wiki/ 实时传输控制协议。

㊁ 参见 http://www.ietf.org/rfc/rfc3551 。

```
5 t=0 0
6 m=audio 21582 RTP/AVP 8 101 13
7 a=rtpmap:101 telephone-event/8000
8 a=fmtp:101 0-16
9 a=ptime:20
```

可以看出，FreeSWITCH 提供的编码类型只有 8（第 6 行，其中 8 代表 PCMA，后面的 101 和 13，分别代表 RFC2833 和静音包支持，不是实际的音频编码），而由于 B 端仅支持 PCMU，所以协商失败，对方返回 488 消息，表示没有找到兼容的媒体类型。在笔者测试中收到的 488 消息内容如下：

```
SIP/2.0 488 Not Acceptable Here
Via: SIP/2.0/UDP 192.168.7.8;rport=5060;branch=z9hG4bKH1t8Fv9Xr7XgK
To: <sip:1002@192.168.7.8:51570;rinstance=59ddd96a3228b9f6>;tag=95be460b
From: "Extension 1000"<sip:1000@192.168.7.8>;tag=QQaKDZ709NHtF
Call-ID: 36dd7df4-6c9d-1231-82ad-cb44f52397c6
CSeq: 46867357 INVITE
User-Agent: Bria 3 release 3.5.0b stamp 69410
Warning: 305 devnull "SDP: Incompatible media format: no common codec"
Content-Length: 0
```

其中的 Warning 是一个可选的消息头，SIP 终端（这里是 Bria）带了这个消息头便于我们查找问题原因。

最后，经过实验笔者发现，如果要使协商成功，只需要在 internal Profile 里将下面一行注释掉：

```
<param name="inbound-zrtp-passthru" value="true"/>
```

需要将该 Profile 重启使之生效，重启命令如下：

```
freeswitch> sofia profile internal restart
```

读者也可以看一下重启前后该配置项的区别（省略其他输出，中文注释为笔者所加）：

```
freeswitch> sofia status profile internal

Name                    internal

ZRTP-PASSTHRU           true                  # 重启前
-------------------------------------------------------------
ZRTP-PASSTHRU           false                 # 重启后
```

尝试再打一个电话，可以看到，这次 FreeSWITCH 向 B 提供的编码列表就多了。下面是 FreeSWITCH 发给 B 的 INVITE 消息中的 SDP 部分：

```
v=0
o=FreeSWITCH 1374381088 1374381089 IN IP4 192.168.7.8
s=FreeSWITCH
c=IN IP4 192.168.7.8
t=0 0
m=audio 26076 RTP/AVP 8 9 0 3 101 13
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=ptime:20
```

其中，“8 9 0 3”分别代表PCMA、G722、PCMU、GSM，由于A提供的编码是PCMA，而FreeSWITCH提供的列表是G722、PCMU、PCMA、GSM，基于FreeSWITCH默认的协商策略（generous），FreeSWITCH在向B发送SDP的时候，尊重客户端A的选择，因此将PCMA排在了最前面，最后的列表成了PCMA、G722、PCMU、GSM。这时候B收到INVITE请求后，由于它仅支持PCMU，因此它在后续的200消息中告诉FreeSWITCH，FreeSWITCH将电话接通后，A便向FreeSWITCH发送PCMA编码的RTP媒体流，FreeSWITCH在收到后会自动转换为PCMU格式的媒体流并发送给B。同理，B向FreeSWITCH发送PCMU，FreeSWITCH转成PCMA再发给A。也就是说，FreeSWITCH可以在A与B端编码不同的情况下进行自动编码转换，简称转码。

通过该例子读者应能了解到，FreeSWITCH使用该协商策略的好处是，由于FreeSWITCH做SDP Offer的时候把PCMA放到了最前面，如果B同时也支持PCMA，而且也采取类似FreeSWITCH的协商策略，选用Offer列表中第一个编码，那么最终协商的结果还是PCMA，最终还是不用转码，从而有效节省了服务器的资源。

8.3.3 透传、媒体绕过与媒体代理

一般情况下，RTP媒体流也是经过FreeSWITCH转发的（见图8-1中的虚线部分）。如上节所述，在G729不支持转码的情况下，只要通话的双方都支持G729，FreeSWITCH仍然是可以建立通话的，其中一种方式就是透传（Passthru）。透传是指在不经过转码的情况下，将从一方收到的媒体流原样转给另一方。这种情况下，仍然可以建立通话，但在需要媒体处理的情况下，如在IVR、录音时，就会有些限制。

另一种极端的情况是采用媒体绕过（Bypass Media）技术，即真正的媒体流使用点到点传输，根本不经过FreeSWITCH。即使是FreeSWITCH从来没听说过的编码，只要通话的双方都支持，FreeSWITCH也能正常建立通话，而RTP则通过点对点直接传输，如图8-2中粗实线部分的RTP（2）路径。当然，使用媒体绕过与透传一样，要求通话的双方必须支持同样的媒体编码。

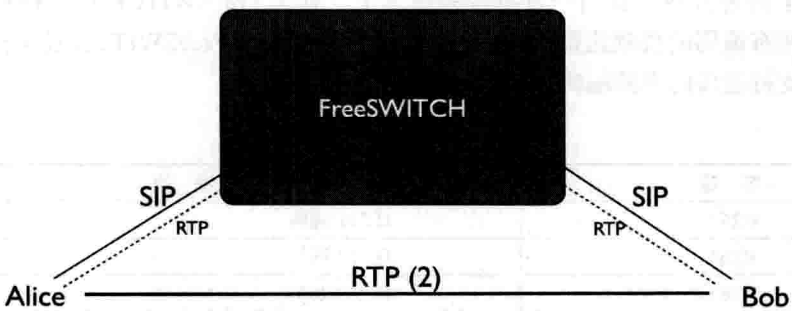


图8-2 Bypass Media

还有一种情况称为媒体代理（Proxy Media），即不管FreeSWITCH是否支持对该种编码

转码，它都对 RTP 数据在不进行任何处理的情况下发给另一方，注意这里它跟透传的区别是，它只改变 SDP 中的“c=”部分，而不对 RTP 进行任何改变，如仍然保留其时间戳等数据。

8.3.4 Media Bug

为了解决监听和录音问题，FreeSWITCH 在媒体流路径上放了一个 Media Bug[⊖]，它的作用相当于我们水管上的一个三通，如图 8-3 所示，媒体流不仅在 A（Alice）和 B（Bob）之间交换，还通过一个三通连接到 C（Carl）。

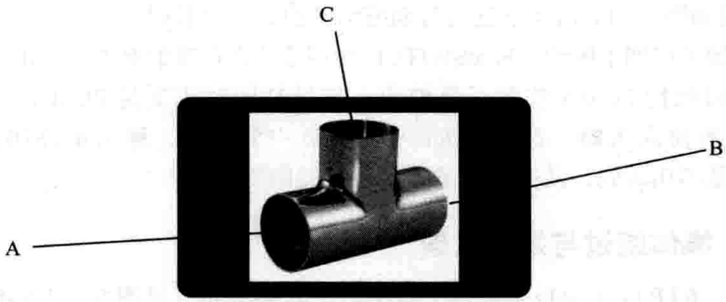


图 8-3 Media Bug 就类似一个三通

技术上讲，C 可能是另一个 SIP Channel，在做监听。Media Bug 的媒体流也是双向的，因而 C 也可以发送音频数据形成三方通话。FreeSWITCH 提供 eavesdrop 和 three_way 两个 App 来分别完成该功能。

实际上，Media Bug 的用处还有很多，如音信号检测（Tone Detect）就是使用 Media Bug 来检测是否有特殊音频的，如传真音或 inband DTMF 信号等。当然，录音也是通过这种机制来实现的（这时候 C 就相当于一个录音机）。

8.3.5 视频

上面为了讲解方便，我们一直拿音频做例子，其实 FreeSWITCH 也支持视频呼叫和会议。目前，所有编码的视频流都是透传的，在本书截稿时，FreeSWITCH 还不支持视频编码的转码。它支持透传的视频编码如表 8-4 所示。

表 8-4 透传的视频编码

名 称	说 明
H261	H.261 视频
H263	H.263 视频
H263-1998	H.263-1998，又称 H263+
H263-2000	H.263-2000，又称 H263++

⊖ 注意这里 Bug 不是我们通常所说的软件的缺陷，而是 Bugging，即监听。

（续）

名　称	说　明
H264	H.264 视频，以上编码均由 mod_h262 支持
Theora	Theora 视频，由 mod_theora 支持
MP4	Mpeg4 视频，由 mod_mp4v 支持
VP8	VP8 视频，由 mod_vp8 支持

下面我们来看一个视频 SDP 的例子。在如下的 SDP 中，可以看到除 m=audio 外，还有一个 m=video，它表示这是一个视频流，IANA 编码 123（注意，它属于大于等于 96 的动态编码）及后面的 rtpmap 属性说明该视频流是 H264 编码的。

```
v=0
o=FreeSWITCH 1364797393 1364797394 IN IP4 192.168.1.102
s=FreeSWITCH
c=IN IP4 192.168.1.102
t=0 0
m=audio 24786 RTP/AVP 8 101
a=rtpmap:8 PCMA/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - - -
a=ptime:20
a=sendrecv
m=video 22252 RTP/AVP 123
a=rtpmap:123 H264/90000
```

视频编码的协商过程与音频类似，不同的是，由于视频不支持转码，因而必须保证两端都使用相同的视频编码才可以互相看到视频。

8.3.6　排错

在实际应用中，媒体的相关的问题可能会较多，各种客户端的实现也五花八门，所以具体的查错和分析方法需要根据当时的情况具体问题具体分析，有一定经验可以帮助你更快地解决问题。这里给出笔者的经验：遇到媒体协商的问题，打个电话并在 Log 中查找与“Audio Codec Compare”相关的行，一般能比较快地定位到问题。

另外，如果在日志中查找不到原因，也可以使用外部的抓包工具（如 tcpdump 或 Wireshark）来抓包分析。关于如何使用这些工具排查错误我们将在后面讲到。这里值得一提的是，FreeSWITCH 也提供了一个实用工具——uuid_debug_media，可以比较方便地查看是否有媒体流的收发，它是一个 API 命令，使用格式如下：

```
uuid_debug_media <uuid><read|write|both><on|off>
```

如以下命令可以在终端上打印当前收到的 RTP 媒体流的信息：

```
uuid_debug_media f87f1d62-ec54-44bd-ab37-cf2fb5b6a5a7 read on
```

另外，在 read/write/both 前面加上 v 就可以打印视频媒体流的信息，如：

```
uuid_debug_media f87f1d62-ec54-44bd-ab37-cf2fb5b6a5a7 vread on
```

8.4 小结

本章我们介绍了与媒体相关的一些基本概念以及各种音 / 视频编码、SDP 及 RTP，也讲了 FreeSWITCH 对媒体处理的方法、协商机制等。理解这些基本概念对于了解整个 SIP 通信的流程和机制都是很有帮助的，熟练掌握和灵活运用这些知识将有助于在以后的使用中快速排查和解决问题。

在实际应用中，可以使用 late-negotiatioin、disable-transcoding 等方法来尽量避免发生转码，以节省系统资源。也可以更进一步设置 Bypass Media，让媒体流不经过 FreeSWITCH 而直接在两个 SIP 客户端间点对点传输，则可以更进一步节约 FreeSWITCH 的资源。当然，在 NAT 环境下，SIP 客户端间的点对点传输可能不可用，那么这时就不能设置 Bypass Media 了。关于具体的应用场景和案例分析，我们将在后面的章节中介绍，这里就不再展开了。

第 9 章 SIP 模块

通过前面的章节我们学习了 SIP 协议的基础知识，并了解到 SIP 协议是现代 VoIP 通信的一个主要协议。事实上，它也是 FreeSWITCH 支持的一个核心协议。FreeSWITCH 中大部分的功能和应用都是基于 SIP 的。因此，在本章我们就来看一下在 FreeSWITCH 中是怎么实现和使用 SIP 的，以及我们如何来配置和使用它。

网络中的 NAT 无处不在，而 SIP 通信的成功与否及质量好坏也与 NAT 息息相关。因此，本章我们也会用一些篇幅介绍 NAT。

9.1 基本概念

本节我们介绍一些与 SIP 相关的基本概念。

- Sofia-SIP ㊀：FreeSWITCH 的 SIP 功能是在 mod_sofia 模块中实现的。FreeSWITCH 并没有自己开发新的 SIP 协议栈，而是使用了比较成熟的开源 SIP 协议栈 Sofia-SIP，以避免“重复发明轮子”㊁。
- Endpoint：在 FreeSWITCH 中，实现一些互联协议接口的模块称为 Endpoint。Free-SWITH 支持很多类型的 Endpoint，如 SIP ㊂、H232 等。这些不同的 Endpoint 主要是

㊀ Sofia-SIP 是由诺基亚公司开发的 SIP 协议栈，它以开源的许可证 LGPL 发布。参见 http://sofia-sip.sourceforge.net/ 。

㊁ 来自英文 Reinvent the Wheel，指重复别人已经建立甚至已经优化了的基本工作和方法，该术语常用于软件开发或其他工程科目中。详见：http://en.wikipedia.org/wiki/Reinventing_the_wheel。当然，选用 Sofia-SIP 的实际原因比这个要复杂。FreeSWITCH 的团队在试验了不下 5 种开源的 SIP 协议栈实现后才最后选定 Sofia-SIP。

㊂ 有的读者可能会问，那么实现 SIP 的模块为什么不支持 mod_sip 呢？这是由于 FreeSWITCH 的 Endpoint 是一个抽象的概念，你可以用任意技术来实现。实际上 mod_sofia 只是对 Sofia-SIP 库的一个黏合和封装。除 Sofia-SIP 外，还有很多开源的 SIP 协议栈，如 pjsip、osip 等。最初选型的时候，FreeSWITCH 的开发团队也对比过许多不同的 SIP 协议栈，最终选用了 Sofia-SIP。FreeSWITCH 是一个高度模块化的结构，如果你喜欢其他协议栈，可以自己实现如 mod_pjsip 或 mod_osip 等，它们是互不影响的。这也正是 FreeSWITCH 架构设计的精巧之处。

使用不同的控制协议跟其他的 Endpoint 通话。所以说，Endpoint 一般是跟通话相关的。

- ❑ mod_sofia：mod_sofia 实现了 SIP 中的注册服务器、重定向服务器、媒体服务器、呈现服务器、SBC 等各种功能。它的定位是一个 B2BUA，它不能实现 SIP 代理服务器[⊖]的功能。
- ❑ SIP Profile：在 mod_sofia 中，有一个概念是 SIPProfile，它相当于一个 SIPUA，通过各种不同的配置参数可以配置一个 UA 的行为。一个系统中可以有多个 SIP Profile，每个 SIP Profile 都可以监听不同的 IP 地址和端口对。
- ❑ Gateway：一个 SIP Profile 中有多个 Gateway，Gateway 可以直译为网关，它主要用于定义一个远端的 SIP 服务器，使 FreeSWITCH 可以与其他服务器通信。FreeSWITCH 可以作为一个 SIP 客户端（UAC）向远端的网关进行注册；当然也可以不注册，而是使用与远端服务器对等的方式（俗称 SIPTrunk，即 SIP 中继）相互通信（我们将在第 14 章讲到 FreeSWITCH 与其他系统相连的各种拓扑结构）。
- ❑ 本地 SIP 用户：FreeSWITCH 可以作为注册服务器，这时候，其他的 SIP 客户端就可以向它注册。FreeSWITCH 将通过用户目录（Directory）中的配置信息对注册用户进行鉴权。这些 SIP 客户端所代表的用户就称为本地 SIP 用户，简称本地用户。
- ❑ 来话和去话：牢记 FreeSWITCH 是一个 B2BUA。如果 Alice 通过 FreeSWITCH 给 Bob 打电话，Alice 首先向 FreeSWITCH 发起呼叫，对 FreeSWITCH 而言，这路通话就称为来话（InboundCall）；然后 FreeSWITCH 再去呼叫 B，这路通话称为去话（OubtoundCall）。如果来、去话都是本地的，又称为本地来话和本地去话。
- ❑ 中继来话或中继去话：如果来、去话不是本地的，双方通话需要以中继方式进行，就称为中继来话或中继去话。但是，中继的叫法只是沿用传统的 PSTN 网络中的概念，在 SIP 术语中，本来是没有中继的概念的。

了解了这些基本概念，下面我们来看一下它们在 mod_sofia 的配置文件中是怎么实现的。

9.2 Sofia 配置文件

Sofia 的配置文件是 conf/autoload_configs/sofia.conf.xml，不过一般不需要直接修改它，因为该文件仅有少数的配置参数，大部分的配置参数实际上是直接在上述配置文件中使用下面的预处理指令装入 conf/sip_profiles/ 目录中的 XML 文件中的配置：

```
<X-PRE-PROCESS cmd="include" data="../sip_profiles/*.xml"/>
```

结合我们在第 5 章学过的配置文件的知识，Sofia 的配置文件的总体结构应该是这样的：

```
<configuration name="sofia.conf" description="sofia Endpoint">
    <global_settings>
        <!-- 全局参数设置 -->
```

⊖ 实现 SIP 代理服务器的开源软件有 OpenSIPS、Kamailio 等。它们可以很好的与 FreeSWITCH 配合工作。

```
    </global_settings>

    <profiles>
        <profile name="profile1">
        </profile>
        <profile name="profile2">
        </profile>
    </profiles>
</configuration>
```

其中，global_settings 中存放了一些全局配置参数。下面的 profiles 标签中又包含多个 profile 标签。每个 profile 标签的详细信息都是在 conf/sip_profiles/ 下的配置文件中配置的。

Sofia 支持多个 Profile，而每个 Profile 相当于一个 SIP UA，在启动后它会监听一个"IP 地址 : 端口"对。读到这里细心的读者或许会发现我们前面有一个不准确的地方——我们在讲 B2BUA 的概念时，实际上只用到了一个 Profile，也就是一个 UA，但我们还是说 FreeSWITCH 启动了两个 UA（一对背靠背的 UA）来为 Alice 和 Bob 服务。从物理上来讲，它确实只是一个 UA，但由于它同时支持多个 Session，在逻辑上就是相当于两个 UA，为了不使读者太纠结于这种概念问题，笔者在前面没有进行太多分析。但到了本章，读者应该非常清楚 FreeSWITCH 中 UA 的含义了——简单来讲，一个"IP 地址 : 端口"对唯一标志一个 UA。

FreeSWITCH 默认的配置带了三个 Profile（也就是三个 UA），在本书中，我们不讨论 IPv6，仅讨论 internal 和 external 两个（它们分别是在 internal.xml 和 external.xml 中定义的）。不要被它们的名字所迷惑，其实 internal 和 external 最大的区别就是一个运行在 5060 端口上，另一个运行在 5080 端口上。有些读者特别关心 internal 和 external 的中文含义（字面意思分别是"内部的"和"外部的"），反而会比较迷惑。当然，两个 Profile 在默认的配置中还有其他区别，我们下面将会讲到。

9.2.1 Profile 配置文件

internel.xml 中定义了一个 Profile（名字就是 internal），它里面有大量的配置参数。这些参数往往与具体的部署方式和呼叫流程有关。到本章为止，我们还没有接触到太多与呼叫流程有关的内容。但是若先讲流程的话，对这里的参数又不熟悉，也不便理解。因此，这就变成了一个是先有鸡还是先有蛋的问题。经过综合考虑，笔者最后决定在本节先把一些重要的配置参数都大体讲一遍，到后面的实战部分在讲到具体流程的时候，涉及这些参数时再深入讨论。

在熟悉了 Sofia 的配置文件结构以后，我们再来看一下 Profile 的配置文件。由于 Sofia Profile 的参数比较多，因此默认把不同的 Profile 配置存放在单独的文件中。

以 internal.xml 为例，首先一个 Profile 的定义是从下面一行开始的：

```
<profile name="internal">
```

该行定义了一个 Profile，它的名字就叫 internal，这个名字本身并没有特殊的意义，也不需要与文件名相同，你可以改成任意你喜欢的名字，但是必须记住它，因为很多地方要使

用这个名字。也可以为 Profile 起个别名⊖，如：

```
<aliases>
    <alias name="default"/>
</aliases>
```

如果有了这个别名，就可以在呼叫字符串中使用这个别名。比如，如果配置文件中有上述配置，就可以使用类似这样的呼叫字符串 sofia/default/13912345678@192.168.1.12 去呼叫相应的 SIP 地址。关于呼叫字符串的知识我们还会在第 12 章讲到。下面我们继续看配置文件。

在一个 Profile 中，可以配置多个网关（Gateway）。网关是在 <gateways></gateways> 标签中定义的。默认的配置中只是装入了相关目录下的网关配置文件。配置如下：

```
<gateways>
    <X-PRE-PROCESS cmd="include" data="internal/*.xml"/>
</gateways>
```

上面的配置代码定义了一些网关。既然 Profile 是一个 UA，它就应该可以注册到别的 SIP 服务器上去，它要注册的远端 SIP 服务器对于该 Profile 来说，就称为网关，而这里的网关配置就是对远端 SIP 服务器的一些描述，即网关的配置参数由远端的 SIP 服务器来决定。我们一般不在 internal 这个 Profile 上使用 Gateway，大多数在 external 上使用（详见 9.3.3 节）。

接下来，便定义该 Profile 所属的域（Domain）。Domain 可以是一个 IP 地址或一个 DNS 域名。需要注意，直接在操作系统 hosts 文件中设置的 IP 到域名的映射可能不好用，如果使用域名，大多数 SIP UA 都需要有一个真正的 DNS 服务器。Domain 的配置代码如下：

```
<domains>
    <!--<domain name="$${domain}" parse="true"/>-->
    <domain name="all" alias="true" parse="false"/>
</domains>
```

在 SIP 配置文件中，Domain 的含义比较容易令人迷惑。因此，一般采用默认的配置就行。不过，总会有些好奇的读者，或者有些高级用户期望了解这些知识，因此我们也简单讲一下。初学者可以自动跳过这一块，以避免“走火入魔”⊖。

一说到 Domain，可能大家都会想到域名。不过在这里，Domain 可以是任意的字符串，可以简单把它理解为一个域。FreeSWITCH 支持多 Domain，也就是逻辑上可以把一些资源（如用户）分到多个“域”中。在本章，我们不讨论多 Domain。感兴趣的读者可以参考 15.7.1 节。

在默认的配置文件中，vars.xml 中定义了一个名为 domain 的全局变量，它的默认值来自于 local_ip_v4 这个全局变量，也就是说，它是一个 IP 地址。当然，话又说回来，domain

---

⊖ 注意，在默认的配置文件中，别名一行是加了注释的，也就是说默认情况下这个别名不起作用，写在那里只是一个例子。此处为了叙述方便，去掉了注释。

⊖ 笔者知道这个词可能不严谨，但实在找不到更恰当的比喻了。

就是一个字符串，它可以是 IP 地址，也可以是其他值，只是默认的配置中是一个 IP 地址。该全局变量在很多地方可以引用，如上面配置例子中的第 2 行（该行默认是注释掉的，因而不生效）。

言归正传。上面配置中第 3 行的含义是：在该 Profile 中定义一个 Domain，name="all" 表示检查看所有的用户目录中定义的 Domain（默认是在 conf/directory 目录下定义的 Domain，如果 name 等于一个特定的 Domain，则它仅会检查指定的 Domain），然后执行下列动作：

- 如果 alias="true"，则会为所有的 Domain 取一个别名，放到与 Profile 同等重要的位置，以便后续在构造或查询呼叫字符串时能找到正确的用户。如在笔者的 FreeSWITCH 中使用 sofia status 命令可以看到如下的行：

```
Name              Type                          Data      State
=================================================================
192.168.1.128     alias                       internal  ALIASED
```

其中，Type 为 alias 表示是一个别名。

- 如果 parse="true"（注意默认的配置中此处为 false），则 FreeSWITCH 会解析该 Profile 中所有的网关。

注意，上面的 alias 和 parse 属性都具有排它性，即在有多个 Profile 的情况下，它们的值只能有一个为 true，否则会产生冲突。

好了，关于这个问题我们就讲到这里，读者可以自行比较一下 internal 和 external 中这一行的区别。另外，我们也将在 15.5.1 节深入讲解 Domain 的实例。

除了上面这些配置外，下面便与该 Profile 相关的一系列参数，大体的配置结构如下：

```
<settings>
    <param name="参数名称" value="参数值">
</settings>
```

我们在下一节再详细讲几个重要的配置参数。

9.2.1 Profile 的几个重要参数

Sofia 的 Profile 有很多可配置参数，它们可以影响某一 Profile 所代表的 SIP UA 的行为。也就是说，对于来话而言，所有到达某一 Profile（如 internal.xml）的呼叫都会进行统一的处理；对于去话而言也类似——所有从某一 Profile 出去的通话也都有类似的行为（如下面将要讲到的 auth-calls 参数）。下面我们就来说一下 Profile 中的一些主要参数，这些参数大部分可以在 internal.xml 找到相应的例子。

inbound-bypass-media 用于设置入局呼叫是否启用“媒体绕过（Bypass Media）”模式，它的取值有“true”和“false”两种。如：

```
<param name="inbound-bypass-media" value="true"/>
```

所有的参数设置格式都差不多，因此，下面我们就不再列出所有代码了。

那么，什么叫 Bypass Media 呢？

如图 9-1 所示，Bob 和 Alice 通过 FreeSWITCH 使用 SIP 接通了电话，他们谈话的语音（或视频）数据要通过 RTP 包传送。RTP 可以像 SIP 一样经过 FreeSWITCH 转发。但是，RTP 与 SIP 相比占用很大的带宽，如果 FreeSWITCH 不需要"偷听（或录音）"他们谈话，为了节省带宽，完全可以让 RTP 直接在两者之间点对点传送，这种情况对 FreeSWITCH 来讲就是没有媒体的，在 FreeSWITCH 中就称为 Bypass Media（媒体绕过）。

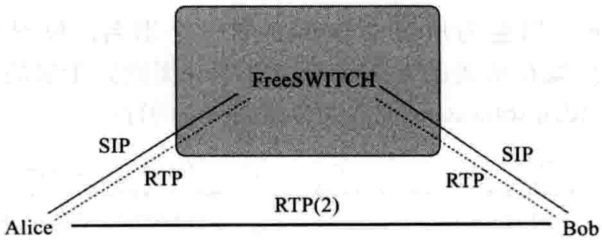


图 9-1 Bypass Media

所以，如果将 inbound-bypass-media 的值设为"true"，那么对于任何来话（如 Alice 打给 Bob，电话要先进入 FreeSWITCH，对 FreeSWITCH 来说该电话就称为来话）都会自动采用 Bypass Media 模式。

下面我们还会看到更多与 Bypass Media 相关的参数。

- media-option：媒体选项，它有两个可能的取值——resume-media-on-hold 和 bypass-media-after-att-xfer。前者的意思是，如果 FreeSWITCH 是没有媒体（No Media/Bypass Media）的，那么若设置了该参数，当你在话机上按下 Hold 键时，FreeSWITCH 将会回到有媒体的状态；后者是跟 Attended Transfer 有关的，Attended Transfe 即出席转移，又称协商转移，它需要媒体配合才能完成工作。但如果在执行转接之前没有媒体（处于 Bypass Media 状态），则该参数能让转接执行时通过 re-INVITE 请求要回媒体，等到转接结束后再回到 Bypass Media 的状态。我们将在 15.1.1 节详细讲解 Attended Transfer。
- context：设置来话将进入 Dialplan 中的哪个 Context 进行路由。

```
<param name="context" value="public"/>
```

需要指出的是，如果用户注册到该 Profile 上（或是经过认证的用户，即本地用户），则用户目录（directory）中设置的 user_contex 优先级要比这里设置的高（见 10.5 节）。

- dialplan：设置默认的 dialplan 类型，即在该 Profile 上有电话呼入后到哪个 Dialplan 中进行路由。

```
<param name="dialplan" value="XML"/>
```

- inbound-codec-prefs：设置支持的来话媒体编码，用于编码协商。默认值引用了 vars.xml 中的一个 global_codec_prefs 变量，该变量的默认值是"G722,PCMU,PCMA,GSM"。

```
<param name="inbound-codec-prefs" value="$${global_codec_prefs}"/>
```

- outbound-codec-prefs：设置去话语音编码。

```
<param name="outbound-codec-prefs" value="$${global_codec_prefs}"/>
```

- rtp-ip：设置 RTP 的 IP 地址。

```
<param name="rtp-ip" value="$${local_ip_v4}"/>
```

- sip-ip：设置 SIP 的 IP 地址。

```
<param name="sip-ip" value="$${local_ip_v4}"/>
```

- sip-port：该 Profile 启动后监听的 SIP 端口号。默认的配置中引用了 var.xml 中定义的一个变量 internal_sip_port，默认是 5060。

```
<param name="sip-port" value="$${internal_sip_port}"/>
```

- auth-calls：设置是否对来电进行鉴权。默认是需要鉴权，即所有从该 Profile 进来的 INVITE 请求都需要经过 Digest 验证。

```
<param name="auth-calls" value="$${internal_auth_calls}"/>
```

- ext-rtp-ip 和 ext-sip-ip：用于设置 NAT 环境中公网的 RTP IP 和 SIP IP。该设置会影响 SDP 中的 IP 地址。

```
<param name="ext-rtp-ip" value="auto-nat"/>
<param name="ext-sip-ip" value="auto-nat"/>
```

其中，value 的取值有以下几种可能：

➢ 一个 IP 地址，如 12.34.56.78；

➢ 一个 STUN 服务器，它会使用 STUN 协议获得公网 IP，如 stun:stun.server.com；

➢ 一个 DNS 名称，如 host:host.server.com；

➢ auto，它会自动检测 IP 地址；

➢ auto-nat，如果路由器支持 NAT-PMP 或 uPnP，则可以使用这些协议获取公网 IP。

这些设置的值可以使用 sofia status 命令显示，如在笔者的电脑上显示结果如下：

```
freeswitch> sofia status profile internal

=================================================================
Name                    internal
RTP-IP                  192.168.1.128
Ext-RTP-IP              119.40.26.181
SIP-IP                  192.168.1.128
Ext-SIP-IP              119.40.26.181
```

我们将在 9.4 节专门介绍 NAT。

本节我们讨论了 SIP Profile 中的几个重要参数，其他的参数在这里就不多解释了。

9.2.2 external.xml

external.xml 是另一个 UA 配置文件，它定义了另一个名为“external”的 UA，默认使

用 5080 端口。从 external.xml 中可以看到，其中的大部分参数都与 internal.xml 中相同。最大的不同是 auth-calls 参数。在 internal.xml 中，auth-calls 默认值是 true ；而在 external.xml 中，默认值是 false。也就是说，客户端发往 FreeSWITCH 的 5060 端口的 SIP 消息需要鉴权（一般只对 REGISTER 和 INVITE 消息进行鉴权），而发往 5080 的消息则不需要鉴权。我们一般把本地用户都注册到 5060 上，所以它们打电话时要经过鉴权，保证只有授权（在我们用户目录中配置的）用户才能注册和拨打电话。而 5080 不同，任何人均可以向该端口发送 SIP INVITE 请求。

如图 9-2 所示，本地用户 1000 注册到 5060 端口上，每次它向外打电话时（呼叫本地用户 1001 时也一样），它向 FreeSWITCH 的 5060 端口发送 INVITE 请求，FreeSWITCH 对其鉴权，验证通过后（可以是 IP 地址验证或 Digest 验证），才允许通话继续进行——如果呼叫 1001，则 FreeSWITCH 继续向用户 1001 发送 INVITE 请求；如果呼叫外部电话，则通过端口 5080 向外部网关发送 INVITE 请求。

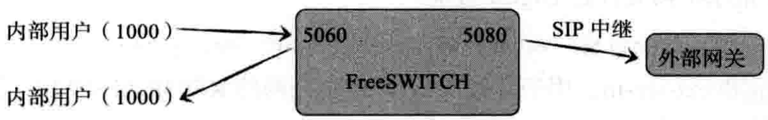


图 9-2 FreeSWITCH 的端口连接

在上面的例子中，FreeSWITCH 也可以通过 5060 端口向外部网关发送 INVITE 请求，效果是一样的。也就是说对于去话（Outbount Call, 出局通话）而言，两者在这里没有区别。但对于来话（Inbound Call, 入局通话）就不同了。为了让 FreeSWITCH 能接收来话，一般有两种方式：

- FreeSWITCH 作为一个 UAC（这里可以理解为软电话，如 X-Lite）从本地的 5080 端口通过 SIP 注册到外部网关上，这样外部网关就可以通过 SIP 消息中的 Contact 字段知道该 UAC 所在的 IP 地址和监听的端口（这里是 5080）。当有电话进来时，外部网关就向 FreeSWITCH 所在 IP 的 5080 端口发送 INVITE 请求。
- 某些网关是把来话和去话分开的，一般不需要或者根本不允许注册。那它们怎么知道我们的 FreeSWITCH 的 IP 和端口呢？在这种情况下，它们一般是在申请的时候事先配置好的，就比方你去装电话时，告诉电信公司你家的门牌号一样，所不同的是这里你告诉人家的是你 FreeSWITCH 服务器的 IP 地址和端口号（在这里是 5080）。如果有来话，外部网关就按照事先配置好的地址发送 INVITE 请求。

不管是使用以上哪种方式，当外部网关的 INVITE 请求到达 FreeSWITCH 时，我们不能要求对这些消息进行鉴权，因为外部网关不是 FreeSWITCH 的本地用户，它们不知道该如何给 FreeSWITCH 提供鉴权信息。所以我们在 external 这个 Profile 中把 auth-calls 参数设为 false，对来话不鉴权。

读到这里，读者大概就明白多了。如果你使用 5060 端口向外部网关进行注册的话，那

么外部网关的来话请求将被发送到 FreeSWITCH 的 5060 端口上，由于 FreeSWITCH 会对发送到该端口的所有来话进行鉴权，很显然外部的网关不知道怎么才能通过 FreeSWITCH 的验证，所以最终 FreeSWITCH 会拒绝所有来话。当然可能你还是有疑问，既然 5080 端口允许所有来话，那么怎么保证安全呢？这个我们将在后面讲安全的时候介绍，在此就不多说了。

9.2.3　Gateway

上一节我们已经提到，FreeSWITCH 需要通过外部网关向外打电话，而这个外部网关就称为 Gateway。在 external.xml 中，我们可以看到它使用预处理指令将 external 目录下的所有的 XML 配置文件都装入到了该 Profile 的 gateways 标签中：

```
<gateways>
    <X-PRE-PROCESS cmd="include" data="external/*.xml"/>
</gateways>
```

这样做的好处是，我们可以把每个网关配置写到不同的文件中。默认的配置中包含了一个 example.xml，里面有很多配置选项。这里我们先从最简单的配置讲起。

如 3.5 节所述，添加一个新网关只需要在 external 目录中新建一个 XML 文件，名字可以随便起，但需要以 .xml 结尾，如“gw1.xml”。其内容如下：

```
<gateway name="gw1">
    <param name="realm" value="SIP 服务器地址 "/>
    <param name="username" value="SIP 用户名 "/>
    <param name="password" value=" 密码 "/>
</gateway>
```

其中，每个网关都有一个名字，由第一行的 name 属性指定。该名字可以与 XML 文件的名字相同，也可以不同。在 FreeSWITCH 内部，将以该名字唯一确定该网关。在整个 FreeSWITCH 中，网关名字必须唯一，否则除第一个外，其他的都将被忽略。

realm 指定 SIP 网关服务器的地址，可以是一个合法的域名或 IP 地址，如果端口号不是 5060，则后面需要加上“:端口号”，如“1.2.3.4:5080”。该参数是可选的，如果没有，则默认跟网关的名字（name）相同（此时网关的名字应该写成一个 IP 地址或者是域名，如 192.168.7.7）。

username 和 password 分别指用户名和密码，这两个参数也是必需的。值得注意的是，有些网关使用 IP 地址验证，而不需要用户名和密码。在 FreeSWITCH 中，你也必须设置这两个参数，但它们的值将被忽略，所以可以填上任意值⊖。

我们来看一下 example.xml 都有哪些默认的参数。注意，默认情况下这些参数都是注释掉的，不起作用。

```
<include>
  <!--<gateway name="asterlink.com">-->
  <!--<param name="username" value="cluecon"/>-->
  <!--<param name="password" value="2007"/>-->
  <!--<param name="realm" value="asterlink.com"/>-->
```

⊖ 这里的 username 将会被填到 SIP 的 From 及其他头域里，所以还是可能会有影响的。

其中，<include> 以及后面的 </include> 标签指明该文件被其他文件包含，它将在预处理阶段被去掉。而且即使没有这两个标签也不会出错。但为了严谨性，建议不要省略这对标签。

下面的代码用于设置 SIP 消息中 From 字段的值，如果省略，则默认与 username 相同。

```
<!--<param name="from-user" value="cluecon"/>-->
```

下面的代码用于设置 From 字段中的 domain 值，默认与 realm 相同。

```
<!--<param name="from-domain" value="asterlink.com"/>-->
```

下面的代码用于设置来话中的分机号，即被叫号码，默认与 username 相同。

```
<!--<param name="extension" value="cluecon"/>-->
```

如果需要代理服务器，则设置该 proxy 的值，默认与 realm 相同。

```
<!--<param name="proxy" value="asterlink.com"/>-->
```

如果需要注册到代理服务器，则设置该 register-proxy 的值，默认与 realm 同。

```
<!--<param name="register-proxy" val” e="mysbc.com"/>-->
```

下面的代码用于设置注册时 SIP 消息中 Expires 字段的值，默认为 3600 秒。

```
<!--<param name="expire-seconds" value="60"/>-->
```

如果网关不需要注册，则设为 false，默认为 true。有些网关必须注册了才能打电话；而有的则不需要。另外，注册到网关上还允许从网关设备呼入我们的 FreeSWITCH。

```
<!--<param name="register" value="false"/>-->
```

下面的代码用于设置 SIP 消息使用 udp 还是 tcp 来承载。

```
<!--<param name="register-transport" value="udp"/>-->
```

下面的代码用于设置如果注册失败或超时，则多少秒后再重新注册。

```
<!--<param name="retry-seconds" value="30"/>-->
```

将主叫号码（要发给对方的）放到 SIP 的 From 字段中。默认会放到 Remote-Party-ID 字段中（有些终端从 From 字段中获取主叫号码）。

```
<!--<param name="caller-id-in-from" value="false"/>-->
```

下面的代码用于设置在 SIP 协议中 Contact 字段中额外的参数。具体的参数需根据实际情况而定，请参考相关的 SIP 协议或对端设备的要求。

```
<!--<param name="contact-params" value="tport=tcp"/>-->
```

每隔一段时间发送一个 SIP OPTIOINS 消息，如果失败，则会从该网关注销，并将其设置为 down 状态。通过周期性地发送无关紧要的 SIP 消息，有助于快速发现对方的状态变化，同时也在 NAT 环境中有助于保持路由器上的 NAT 映射关系，保持连接畅通。

```
<!--<param name="ping" value="25"/>-->
```

最后是网关及 include 的关闭标签，保持 XML 的完整性。

```
<!--</gateway>-->
</include>
```

到此，示例网关配置文件中的参考就都介绍完了。这些参数在与其他设备对接时比较有用。FreeSWITCH 的兼容性很强，通过调整 Profile 及 Gateway 的参数，可以兼容已知的绝大部分 SIP 设备。

9.3 常用命令

mod_sofia 提供了一个 API 命令——sofia，它有很多参数，可以提供很多功能。比如，我们熟悉的 sofia status 会列出 sofia 的运行状态。在 FreeSWITCH 控制台上输入 sofia 或 sofia help 命令，将得到命令的帮助信息。帮助信息如下：

```
freeswitch> sofia

USAGE:
--------------------------------------------------------------------------------
sofia global siptrace <on|off>
sofia        capture  <on|off>
            watchdog <on|off>

sofia profile <name> [start | stop | restart | rescan] [wait]
                     flush_inbound_reg [<call_id> | <[user]@domain>] [reboot]
                     check_sync [<call_id> | <[user]@domain>]
                     [register | unregister] [<gateway name> | all]
                     killgw <gateway name>
                     [stun-auto-disable | stun-enabled] [true | false]]
                     siptrace <on|off>
                     capture  <on|off>
                     watchdog <on|off>

sofia <status|xmlstatus> profile <name> [reg [<contact str>]] |
                                        [pres <pres str>] |
                                        [user <user@domain>]
sofia <status|xmlstatus> gateway <name>

sofia loglevel <all|default|tport|iptsec|nea|nta|nth_client|nth_server|nua|soa|
sresolv|stun> [0-9]
sofia tracelevel <console|alert|crit|err|warning|notice|info|debug>

sofia help
```

下面，我们对主要的参数进行讲解。

9.3.1 状态相关命令

我们已经很熟悉 sofia status 命令了，其可以列出当前 mod_sofia 的运行状态，其还有如下功能：

❑ 列出某个 Profile 的状态。

```
freeswitch> sofia status profile internal
```

❑ 列出某个 Profile 上所有已注册用户。

```
freeswitch> sofia status profile internal reg
```

❑ 过滤某些符合条件的用户。

```
freeswitch> sofia status profile internal reg 1000
```

❑ 列出某个特定用户。

```
freeswitch> sofia status profile internal user 1000
```

❑ 列出网关状态。

```
freeswitch> sofia status gateway gw1
```

以上的命令都可以将 status 用 xmlstatus 来替代，以列出 XML 格式的状态，这样比较容易用其他程序解析，读者可以自行比较它与“sofia status”命令输出的异同。

9.3.2 Profile 相关命令

Profile 相关的命令都是针对某个 Profile 进行的操作。一般来说，下面这些指令在需要读取 XML 时都会隐含 reloadxml，因而如果要修改 XML 配置文件中的某个参数，就不需要明确的 reloadxml 指令了。

常用的启动、停止、重启某个 Profile 的命令分别如下：

```
freeswitch> sofia profile internal start              # 启动
freeswitch> sofia profile internal stop               # 停止
freeswitch> sofia profile internal restart            # 重启
```

有时候，修改了某个 Profile 的某个参数（如 outbound-codec-prefs、inbound-late-negotiation 等），不需要重启（因为重启是影响通话的）。可以使用下列命令让 FreeSWITCH 重读 sofia 的配置（并不是所有的配置参数都能生效）：

```
freeswitch> sofia profile internal rescan
```

添加了一个新的 gateway 以后，也可以使用上述 rescan 指令读取。

如果是修改了一个网关，则可以先将该网关删除，再 rescan，如：

```
freeswitch> sofia profile external killgw gw1          # 删除一个网关
freeswitch> sofia profile external rescan              # 重读参数
```

下列命令可以指定某个网关立即向外注册或注销：

```
freeswitch> sofia profile external register gw1        # 注册
freeswitch> sofia profile external unregister gw2      # 注销
```

开启该 Profile 的 SIP 跟踪功能抓 SIP 包：

```
freeswitch> sofia profile internal siptrace on
```

有时候，希望将已经注册的用户清理掉。可以使用如下方法实现，如：

```
freeswitch> sofia profile internal flush_inbound_reg 1000@192.168.1.7
```

也可以通过找到该用户的 call-id 来清理，如下面的例子，再重新查看状态时，发现已

经被清除了（Total items 变成了 0）：

```
freeswitch> sofia status profile internal reg 1000

Registrations:
================================================================================
Call-ID:          ZWIyNDVmMTU1ODIyMDA1ZDl
User:             1000@192.168.1.123
...

Total items returned: 1

freeswitch> sofia profile internal flush_inbound_reg ZWIyNDVmMTU1ODIyMDA1ZDl

+OK flushing all registrations matching specified call_id

freeswitch> sofia status profile internal reg 1000

Total items returned: 0
```

有的时候读者可能会发现记录根本没有清除，出现这种情况可能是因为输错了命令或参数，但更可能是因为记录已经清除了，但客户端又重新注册了（因而又产生了一条新记录）。在 flush_inbound _reg 时，FreeSWITCH 只是简单地清理本地数据库中用户的注册信息，它无法防止客户端重新注册。如果确实不想让该客户端再注册了，可以给该用户改一个密码，让它注册不上来，或直接修改 iptables 防火墙，不允许相关的 IP 注册。

9.3.3 SIP Capture

FreeSWITCH 内置了 Homer Capture Agent 用于 SIP 抓包。Homer[⊖]是一个使用 HEP、HEP2 和 IPIP 协议的抓包分析工具。它逻辑上由 Capture Agent、Capture Node（又称 Capture Server）和 webHomer 三部分组成。其中，Captuer Agent 运行于 FreeSWITCH 内部，用于将收到的 SIP 包进行封装并通过 HEP/HEP2 或 IPIP 协议发送到远端的 Capture Node 上。Capture Node 收到封装后的 SIP 包以后，进行分析并将分析结果存储到数据库中。然后，技术人员就可以使用 webHomer 在浏览器中查看各种统计和分析结果了，如图 9-3 所示。

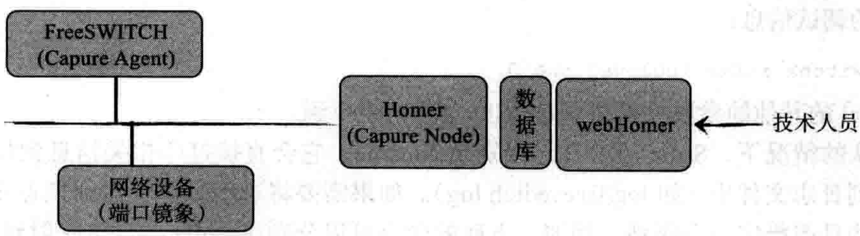


图 9-3　Homer 示意图

⊖ 参见 http://www.sipcapture.org/。

使用 Capture 功能前，需要先在 sofia.conf.xml 中配置 Captcher Node 的地址，如：

```
<param name="capture-server" value="udp:192.168.0.100:6060"/>
```

通过在 Profile 中配置下列参数，可以配置该功能默认情况下在 Profile 加载时是打开的还是关闭的，如：

```
<param name="sip-capture" value="yes"/>
```

当然，也可以在运行阶段通过命令动态开关，如：

```
freeswitch> sofia profile internal capture on
freeswitch> sofia profile internal capture off
```

9.3.4 global 相关

在使用 sofia 命令的 siptrace 子命令进行抓包时，用户经常搞不清该对哪个 Profile 进行抓包。因此，FreeSWITCH 的作者为这部分用户加入了这个特性——通过使用 global 参数，使 siptrace 子命令对所有 Profile 都有效，如以下两条命令可以分别打开和关闭全局的 SIP 消息跟踪：

```
freeswitch> sofia global siptrace on
freeswitch> sofia global siptrace off
```

以下两条命令可以分别打开和关闭全局的 SIP 捕获（Homer 方式抓包）：

```
freeswitch> sofia global captuer on
freeswitch> sofia global capture off
```

9.3.5 debug 相关

有时候，可能是协议栈更底层的原因引起的问题，由于收到或发送非法的消息会导致协议栈出错，这可能会使消息丢弃，当然也可能是协议栈层的 Bug，在这种情况下即使开启了详细的 FreeSWITCH 日志以及 SIP 跟踪（siptrace）也查找不到问题的原因。这时候可以使用如下命令打开更低级别的调试器：

```
freeswitch> sofia loglevel all 9
```

以上命令将开启详细的 Sofia SIP 底层调试信息，在控制台上打印日志。其中日志级别为 0 到 9。如果你对 sofia 比较熟悉，也可以开启相关模块的日志。例如，如下命令可以仅开启 nua 的调试信息：

```
freeswitch> sofia loglevel nua 9
```

loglevel 的其他的参数可以在 sofia 的命令帮助中找到。

在默认的情况下，Sofia 层的日志级别是 console，它会直接打印相关信息到控制台上，而不会写到日志文件中（如 log/freeswitch.log）。如果需要将这些日志也写到日志文件中去，可以为这些日志指定一个级别。例如，下面的命令可以分别将 Sofia 的日志映射到 debug 和 notice 级别：

```
freeswitch> sofia tracelevel debug
freeswitch> sofia tracelevel notice
```

最后，可以使用如下命令关闭这些调试：

```
freeswitch> sofia loglevel all 0
```

9.3.6 其他命令

除 sofia 外，mod_sofia 还提供了一些其他 API 命令，这些命令一般用于显示一些相关的信息，简介如下。

- sofia_username_of。返回注册用户的 username。

```
freeswitch> sofia_username_of 1000@192.168.1.123

1000
```

- sofia_contact。返回注册用户的联系地址。

```
freeswitch> sofia_contact 1000@192.168.1.123

sofia/internal/sip:1000@192.168.1.123:39988;rinstance=e9ce1bbdd471ab2d
```

- sofia_count_reg。在允许多点注册的情况下（开启 multiple-registrations 时），计算有多少客户端注册了。

```
freeswitch> sofia_count_reg 1000@192.168.1.123

1
```

- sofia_dig。类似于 DNS 的 dig，返回其他服务器的服务地址和端口号，如下命令显示了指定 IP 上都有哪些 SIP 服务。

```
freeswitch> sofia_dig 192.168.1.123

Preference        Weight    Transport        Port      Address
================================================================================
          1         1.000          udp       5060  192.168.1.123
          2         1.000          tcp       5060  192.168.1.123
```

其中，192.168.1.123 上有两个 SIP 服务，它们分别监听 udp 和 tcp 的 5060 端口。

- sofia_presence_data。显示 Presence 数据，下面的命令显示 1000 处于 Busy 状态：

```
freeswitch> sofia_presence_data status 1000@192.168.1.123

Busy
```

下面的命令列出指定用户的 Presence 信息：

```
freeswitch> sofia_presence_data list 1000@192.168.1.123

status|rpid|user_agent|network_ip|network_port
Busy|busy|Bria 3 release 3.5.0b stamp 69410|192.168.1.123|46342
+OK
```

下面的命令可以列出用户的 user_agent 信息：

```
freeswitch> sofia_presence_data user_agent 1000@192.168.1.123
```

```
Bria 3 release 3.5.0b stamp 69410
```

9.3.7 其他

笔者经常被问到的一个问题是——Sofia Profile 的参数众多，修改哪些参数需要重启，哪些不需要？这个问题现在还没有统一的答案。一个基本原则是，能不重启就不重启，如修改上面所说的 inbound-codec-prefs 或 outbound-codec-prefs 等。但有一些参数，如跟 IP 地址和端口号相关的参数，则一般需要重启，重启 Profile 前需先中断现有通话。

9.4 NAT 穿越

默认安装的 FreeSWITCH 也能很好地工作在 NAT 环境下，但不可否认，用户的网络环境可能五花八门，因而也不可避免地会遇到 NAT 问题。适当地解决在 NAT 网络环境下的内、外网通信问题，就称为 NAT 穿越。

实际上，SIP/RTP 协议本身的特性决定了它与 NAT 的各种恩怨。NAT 涉及的知识比较多，用户不仅要了解 FreeSWITCH、SIP 协议及跟踪调试技巧，还要了解网络拓扑结构、使用网络设备（如交换机和路由器）的相关参数和特性等。因而对于初学者而言，笔者不建议大家花费时间去研究和解决 NAT 问题，而是建议在局域网环境中学习 FreeSWITCH，等有了一定基础以后再研究 NAT 环境。

当然 NAT 问题也不是那么可怕，若初学者有兴趣也不妨读一下本节的内容，即使不为解决 NAT 的问题，对理解 SIP 协议以及本节前面讲到的参数配置也是有帮助的。另外，笔者把这部分内容放到本章是也因为它跟 mod_sofia 模块结合比较紧密。

好了，言归正传。NAT 的全称是 Network Address Translation（网络地址转换），它最初是为了克服 IPv4 网络地址不足出现的一项技术。众所周知，IPv4 使用 32 位（4 个 8 位，即 4 字节）地址空间，因而最多可以表示 2^{32} 个地址。随着 Internet 的迅速发展，人们需要大量的 IP 地址，大大超出了 IPv4 所能提供的地址数量。为了解决 IPv4 地址空间紧张的问题，RFC1918 ⊖规定了一些私有的 IP 地址，这些私有的地址存在于路由器后面，它们本身形成一个私有的网络，称为内部网（Intranet，简称内网）。当内网的 IP 需要与外界的 IP（又称公网）通信时，通过路由器提供的网络地址转换（NAT）功能转换成一个合法的外网 IP 地址来与外界通信。不同的内部网间彼此独立，因而可以复用这些私有的 IP 地址，这大大提高了 IPv4 网络的接入能力。

虽然 NAT 有效解决了 IPv4 网络上的 IP 地址短缺问题，但对于飞速发展的互联网来说，还不是终极的解决方案。因此，人们很早就发明了 IPv6，它使用 128 位的地址空间，能表示 2^{128} 个地址，也就是说将来任何可以联网的智能设备，包括家用的微博炉、手表等都可以获取一个 IPv6 地址。为了推动 IPv4 向 IPv6 的过渡，国际 IP 地址分配机构

⊖ http://tools.ietf.org/html/rfc1918 。它规定的私有 IP 地址有：A 类：10.x.x.x；B 类：172.16.x.x ~ 172.31.x.x；C 类：192.168.x.x。

ICANN ⊖早在几年前就提出了 IPv4 地址预警，称地址将很快耗尽，但遗憾的是，这项工程还是迟迟没有进展。Anthony Menissale《 FreeSWITCH 1.2》中说过："世界改变得越快，人们就越容易怀旧。技术的进步和革命也是如此。我们的汽车仍然假装有速度指针，你喜欢的网站上仍然使用过时的按钮和开关图片，我们作为社会的一分子，仍然信奉一句话：东西，如果它不坏，就不要去修它" ⊜。

所以，我们还是整天生活在 NAT 的环境中，不管是在办公室，还是在家里通过路由器上网时。大多数人并未意识到 NAT 的存在，因为大多数的 Web 应用都是基于 TCP 的，NAT 对 TCP 的影响不是很大。但基于 SIP 的语音通信中，SIP 一般用 UDP 承载，UDP 是无连接的协议，因而在 NAT 穿越方面就更加困难。而更为复杂的是，媒体（语音或视频）数据是在 RTP 包中传递的，它是与 SIP 路径不同的另一路径（甚至是更多的路径，比如同时有语音和视频的情况）。虽然 SIP 也可以通过 TCP 承载，但 RTP 为了保持实时性，还需要用 UDP 传输。因而，我们就不能忽略 NAT 的影响了。

图 9-4 所示为一个典型的 NAT 结构。其中，路由器有两个网络接口，一个用于连接外网，其 IP 是 1.2.3.4，一个用于连接内网，其 IP 是 192.168.0.1。内网的主机 IP 从 192.168.0.2 到 192.168.0.5，它们把 192.168.0.1 作为一个网关，即所有与外网的通信都需要经过 192.168.0.1 转发。如果内网主机要与外界通信，路由器会将内网主机的请求转换成外网的 IP 地址，因而不管是哪个内网的主机与外界通信，外界的主机看起来都是从 1.2.3.4 这个路由器的外网 IP 发出的。同时，路由器会维护一个地址与内网主机间的映射关系，以保证回来的 IP 能到达相应的主机。这个映射关系是在内网主机首次向外网发包时建立的，此后外网的主机才可以向内网的主机发送信息。建立该映射关系的过程好像是在 NAT 设备上打了一个"洞"(因而该技术也称为 UDP Hole Punching，即打洞)，通过该"洞"进行内外网的通信。该洞是有生命周期的，如果在一段时间内没有数据通过，则洞会自动消失。

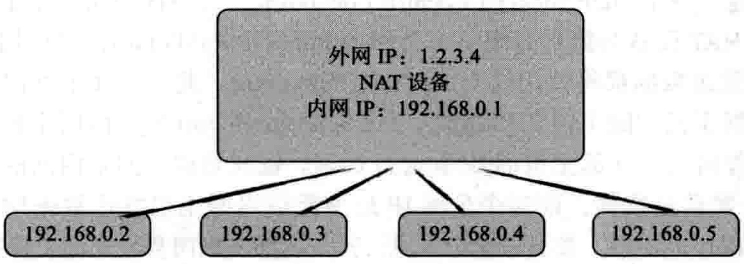


图 9-4 NAT 示意图

⊖ CANN 全称为 The Internet Corporation for Assigned Names and Numbers，即互联网名称与数字地址分配机构。

⊜ The more things change, the more everybody wants it to stay the same. It's just a part of how technology evolution works. Our cars still pretend to have speedometer needles; the graphics on your favorite website look like an old-fashioned set of buttons and switches. We as a society also try to live by the motto: "If it ain't broke, don't fix it!"

9.4.1 NAT 的种类

NAT 有三种类型：静态 NAT（Static NAT）、动态地址 NAT（Pooled NAT）和网络地址端口转换（Network Address Port Translation，NAPT）。其中静态 NAT 设置起来最简单；内部网络中的每个主机都被永久映射成外部网络中的某个合法的地址，而动态地址 NAT 则是在外部网络中定义了的一系列的合法地址，采用动态分配的方法映射到内部网络；NAPT 则是把内部地址映射到外部网络的一个 IP 地址的不同端口上。

前两种类型实际上都不能节省 IP 地址，我们就不讨论了。NAPT 是人们比较熟悉的一种转换方式，它普遍应用于接入设备中，可以将中小型的网络隐藏在一个合法的 IP 地址后面，这个优点使得这种方式在小型办公室内非常实用，通过从 ISP 处申请的一个公网 IP 地址，可以将多台电脑或网络设备通过 NAPT 接入 Internet。

一般来说，NAPT 有四种类型，但它实际上又分为两大类：锥型 NAT 和对称 NAT。其中锥型 NAT 又分为三类，因而一共是四类。简单起见，我们沿用四类的说法来简单讲解（其中前三类属于锥型，最后一类属于对称型）。

（1）Full Cone NAT（全锥型 NAT）

内网主机建立一个 UDP socket（表示为 LocalIP:LocalPort，如 192.168.0.2:5060），第一次使用这个 socket 给外部主机发送数据时，NAT 设备（如我们上面讲的路由器）会给其分配一个公网 IP、端口对（PublicIP:PublicPort，如 1.2.3.4:5060），并记住它们之间的映射关系。以后用这个 socket 向外面任何主机发送数据都将使用这对 PublicIP:PublicPort。此外，任何主机只要知道这个 PublicIP:PublicPort 就可以给它发送数据，NAT 设备会根据它已经记住的映射关系将收到的数据转发到相应的内部主机的 LocalIP:LocalPort 上。其实简单来说就一句话：**内部主机向外打了一个洞，外网的任何主机都可以利用这个洞与它通信**。

（2）Restricted Cone NAT（限制锥型 NAT）

内网主机建立一个 UDP socket（LocalIP:LocalPort），第一次使用这个 socket 给外部主机发送数据时 NAT 设备会给其分配一个公网的 PublicIP:PublicPort，以后用这个 socket 向外面任何主机发送数据都将使用这对 PublicIP:PublicPort。此外，如果外部主机想要发送数据给这个内网主机，除了需要知道这个 PublicIP:PublicPort 外，内网主机在这之前必须用这个 socket 曾向这个外部主机的 IP 发送过数据。也就是说，如果内网的主机从来没有往某一公网 IP 发送过数据，则这个公网 IP 是不能往内网主机发送数据的，即使它知道 PublicIP:PublicPort 也不行。按洞的理论来说，就是**内部主机向某一外部主机打了一个洞后，只有该外部主机才能利用这个洞**。

（3）Port Restricted Cone NAT（端口限制锥型 NAT）

这种 NAT 与 Restricted Cone 类似，唯一不同的是，如果外部主机想要给内网主机发送数据，它除了必须知道 PublicIP:PublicPort 外，而且内部的主机必须事先向该外部主机的 IP:Port 发送过数据，并且该公网主机必须使用相应的 IP:Port 通过 PublicIP:PublicPort 给内网主机发送数据。

可以看出，Poxt Restricted Cone NAT 与 Restricted Cone 相比增加了对外网主机使用的

端口的限制。**即内部主机向外部主机上一某一程序（一个端口）打了一个洞，则只有该程序可以利用这个洞，其他的不行。**

（4）Symmetric NAT（对称型 NAT）

锥型 NAT 对内网主机的同一 socket（LocalIP:Localport）发往任何外网主机的数据均分配一个 PublicIP:PublicPort，而对称型 NAT 会对内网主机同一 socket（LocalIP:Localport）发往外部不同主机的数据分配不同的 PublicIP:PublicPort 映射关系。

详细来讲，内网主机建立一个 UDP socket（LocalIP:LocalPort），当用这个 socket 第一次给外部主机 1 发送数据时，NAT 设备会为其映射一个 PublicIP-1:Port-1，以后内网主机发送给外部主机 1 的所有数据都用这个 PublicIP-1:PublicPort-1。如果内网主机同时用这个 socket 给外部主机 B 发送数据，第一次发送时，NAT 设备会为其分配一个 PublicIP-2:PublicPort-2，以后内网主机发送给外部主机 2 的所有数据都用这个 PublicIP-2:Port-2。

如果任何外部主机 M 想要发送数据给这个内网主机 N，N 必须曾经用这个 socket 向这个外部主机 M 的 IP 发送过数据，并且 M 需要知道 N 向 M 发送数据时 NAT 设备为其映射的 PublicIP:PublicPort。这样这个外部主机 M 就可以用自己的 IP：AnyPort（即任何端口）给 PublicIP:PublicPort 发送数据。

对称型 NAT 相当于对同一内部主机联系不同的外部主机时都需要打不同的洞。

9.4.2 FreeSWITCH 的拓扑结构

FreeSWITCH 一般有三种拓扑结构，如图 9-5 所示。

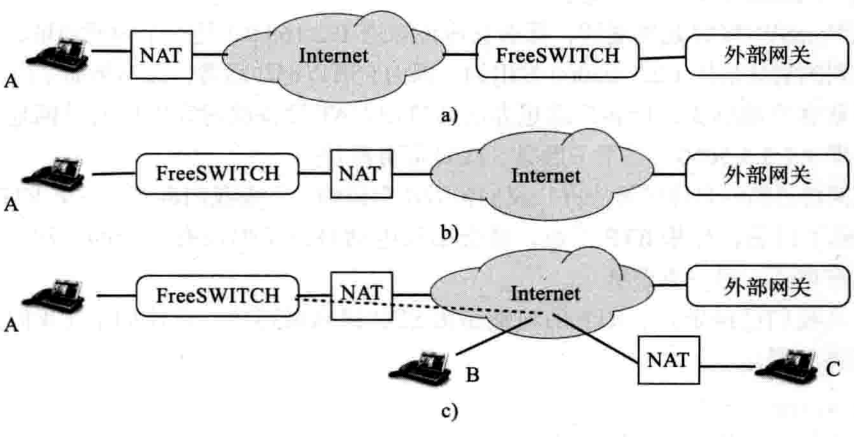


图 9-5 常见 FreeSWITCH NAT 网络拓扑结构

如图 9-5a 所示，FreeSWITCH 运行在公网上，与公网上的其他落地网关设备对接。SIP 话机一般位于 NAT 内部。这种情况一般用于运营 VoIP 业务的情况。

如图 9-5b 所示，FreeSWITCH 运行在内网上，穿过 NAT 与公网上的设备对接。SIP 话机也在内网上。一般用于公司内部的 IP-PBX 的情况。

如图 9-5c 所示，与图 9-5b 所示不同的是，公网上的用户（如 B）也可以穿过 NAT 向 FreeSWITCH 注册，或通过 FreeSWITCH 打电话，而其外网用户通常是公司的外勤或出差人员。这种拓扑结构通常比较复杂，尤其是在希望 B 能在内网和外网间自由切换的情况下（有时内勤有时外勤，但不希望电脑上的 SIP 软电话注册地址改来改去）。甚至处于另外一个 NAT 后面的用户（如 C）也希望向 FreeSWITCH 注册，这种情况称为双重 NAT（Double Nat)，就更加复杂了。

9.4.3 NAT 是怎么影响 SIP/RTP 通信的

我们来看图 9-5a 所示的拓扑结构（下面除非特别说明我们都以这种拓扑结构为例）。假设 SIP 话机的 IP 地址是 192.168.0.2，而路由器的外网 IP 是 1.2.3.4，FreeSWITCH 的 IP 地址是 1.2.3.5。如果 SIP 话机向 FreeSWITCH 注册，它将发送以下 REGISTER 消息（省略了其他不需要的字段）：

```
REGISTER 1000@1.2.3.5 SIP/2.0
Contact: 1000@192.168.0.2:5060
```

该消息是从 192.168.0.2:5060 发出的，由于 NAT 设备进行了网络地址转换，因此 FreeSWITCH 在收到请求后会认为该消息是从 1.2.3.4:5060 发出的[⊖]。

大家都已经知道，SIP 话机向 FreeSWITCH 注册是为了让 FreeSWITCH 记住自己的 Contact 地址。但如果 FreeSWITCH 记住了 192.168.0.2，由于它没法连通该地址，因此如果有人呼叫 1000 这个用户就会出现电话打不通的情况，这就是典型的 NAT 引起的问题。

解决这一问题通常有两种思路：

- 如果 FreeSWITCH 足够聪明，那么它应该知道 192.168.0.2 是一个内网地址，并且由于它收到的消息是从 1.2.3.4:5060 发出的，因而它可以记住后者，而不是那个内网地址。
- 可以从客户端解决，让客户端想办法知道被 NAT 设备映射完以后的外网地址和端口号，即 1.2.3.4:5060。这个一般靠 STUN 服务解决。

针对这两种思路的具体的解决办法我们稍后还会讲到。下面我们再来看一下 RTP 的情况。

SIP 走通了以后，如果 RTP 不通，就会出现电话打通了但没有声音的情况（或者是单通，即一方有声音，另一方没有）。

在第 7 章我们已经学过，RTP 的地址是由 SDP 信息描述的，具体如下（我们照样省略了其他不相关内容）：

```
INVITE 9196@1.2.3.5
Content-Type: application/sdp

c=IN IP4 192.168.0.2
m=audio 50452 RTP/AVP 8 0 101
```

跟 SIP 消息类似，FreeSWITCH 是无法向这个 192.168.0.2 发送 RTP 包的，这个问题可以由客户端通过 STUN 服务解决，也可以由 FreeSWITCH 端解决。如 FreeSWITCH 可以在

⊖ 路由器映射的端口号也可能不是 5060，比如，如果有两个话机同时注册，另一个映射可能成 5061。

收到第一个 RTP 包时得知该 RTP 流对应的外网 IP 和端口号（如 1.2.3.4:50452），以后所有的 RTP 流都发送到这个 NAT 设备的外网地址上，NAT 设备会将收到的从 FreeSWITCH 发送过来的 RTP 包转发到该内网地址上，话机就能“听”到声音了。

上面我们只谈了图 9-5a 所示的拓扑结构，在图 9-5 所示的另外两种拓扑结构下，FreeSWITCH 就相当于这里说的 SIP 话机，它跟外网设备通信时就需要通过 STUN 之类的服务来获取相应的外网 IP 和端口号。

9.4.4 NAT 的穿越方法

通过对 9.4.3 节的学习，我们知道要解决 NAT 穿越问题就要解决内、外网地址映射的问题。也就是说，除了 NAT 设备自己知道这个映射关系以外，SIP 客户端（如话机）和服务器（如 FreeSWITCH）都需要知道。

在继续尝试解决 NAT 穿越问题前，我们先做以下的准备：

- 了解你的网络环境和拓扑结构。如了解自己的 IP 地址段、网关 IP、外网 IP、使用的路由设备厂商和型号、提供接入的服务商或运营商等。另外有一些在线工具可以帮你快速知道自己的外网地址，如你可以直接访问国内的 http://ip138.com/ 或国外的 http://ifconfig.me。笔者经常用下列命令获取自己的外网地址：

```
$ curl ifconfig.me

119.40.26.181
```

- 禁用路由器的 ALG（Application Level Gateway）。某些路由器有 ALG 功能，它们会修改 SIP 包中的 IP 地址，“帮助”你进行 NAT 穿越。但很遗憾的是，它们实现的往往有 Bug，而且该功能默认是打开的。FreeSWITCH 不需要 ALG。

实际上，NAT 问题本来是不需要 FreeSWITCH 来解决的。在理想的情况下，任何可能用于 NAT 后端的设备要想与外界通信，都必须能自己解决 NAT 穿越问题。但事实上，现实世界不是理想的国度，很多设备都无法解决 NAT 穿越问题。因此 FreeSWITCH 团队通过深入研究，提供了很多方法来帮助解决 NAT 的穿越问题[1]。这些方法在默认的安装下一般都能工作得很好，但在不一般的情况下还是需要手工处理的[2]。

---

[1] To be honest, the original stance of the FreeSWITCH developers on the NAT issue was, "Not our problem!" In an ideal world, every device behind a NAT firewall is well aware of its circumstances and can successfully solve its own problems. Unfortunately, we do not live in an ideal world (of course if we were in an ideal world, none of us would have a job because there would be no problems to solve). So we decided, "OK fine, we'll give it a shot!" We soon learned that our users had a myriad of devices that they wanted to use with FreeSWITCH, but these devices had absolutely no idea how to deal with NAT. Soon, we began the monumental task of developing techniques to allow these devices to work despite their shortcomings. NAT is a vicious opponent and the faint-hearted do not stand a chance to survive.——摘自《FreeSWITCH 1.2》

[2] Anthony 也说过：“如果你觉得 NAT 问题这么令人困惑，请自我安慰一下，实际上我们已经尽力把这个问题简化了。如果现在你还是觉得它很复杂，实际情况是它以前更差。”原文是：“If you find this whole thing to be completely confusing, take solace in the fact that we have actually simplified it for you. So while it may seem to be crazy right now, the fact is that it was much worse before.”——摘自《FreeSWITCH 1.2》

1. SIP 穿越

FreeSWITCH 默认使用 ACL 来判断对方是否处于一个 NAT 环境中，配置项如下：

```
<param name="apply-nat-acl" value="nat.auto"/>
```

其中，nat.auto 是一个 ACL，它包含 RFC1918 规定的私网地址，并去掉了本地网络的地址。当 SIP 客户端向 FreeSWITCH 注册时，FreeSWITCH 会比较 SIP 消息中的 Contact 地址是否包含在这个 ACL 中，如果包含，说明是来自一个 NAT 背后的设备，那么它就把其 Contact 地址自动替换为与该设备对应的外网地址（即 SIP 包的来源地址，已经被 NAT 设备转换成了外网地址），因而接下来再有人呼叫它时，FreeSWITCH 就能正常给它发 INVITE 请求了。

下列命令列出了一个 SIP 客户端通过 NAT 注册的情况（其中 Contact 中进行了人工换行）：

```
recv 557 bytes from udp/[1.2.3.4]:56020 at 04:05:37.636135:
----------------------------------------------------------------------
REGISTER sip:1.2.3.5 SIP/2.0
Contact: <sip:1002@192.168.0.2:56020;rinstance=6622cfa07990b53c>
To: "1002"<sip:1002@1.2.3.5>
From: "1002"<sip:1002@1.2.3.4>;tag=7ae09a77
Call-ID: YjU0OTdhYmRmZTEyZGU5MjM5ZGEyY2I0ZjY0NDhjZDk
User-Agent: Bria 3 release 3.5.0b stamp 69410
```

注册成功后，可以在 FreeSWITCH 中查看其注册情况：

```
freeswitch> sofia status profile internal reg
Registrations:
==========================================================================
Call-ID:        YTU5MzZiMjBlODkwNzQ1NGQ1Zjg0OTE3YTUxNGExNWI
User:           1002@1.2.3.5
Contact:        "1002" <sip:1002@192.168.0.2:54892;
                rinstance=1d165ab726aed220;fs_nat=yes;
                fs_path=sip%3A1002%401.2.3.4%3A54892%3B
                rinstance%3D1d165ab726aed220>
Agent:          Bria 3 release 3.5.0b stamp 69410
Status:         Registered(UDP-NAT)(unknown) EXP(2013-08-17 12:51:10) EXPSECS(3658)
```

可以看到，其中 Contact 字段中有 fs_nat=yes 标志，它表示 FreeSWITCH 已经“聪明”地知道了该客户端是处于一个 NAT 后面。上面的显示是经过 urlencode 的，不直观，我们用 urldecode ⊖反编码如下：

```
sip:1002@192.168.0.2:54892;rinstance=1d165ab726aed220;
fs_nat=yes;fs_path=sip:1002@1.2.3.4:54892;rinstance=1d165ab726aed220
```

从上面的信息可以看到，FreeSWITCH 既知道它的内网地址，又知道它的外网地址。以后任何时候 FreeSWITCH 向这个客户端发消息时，均使用 fs_path 指定的地址。

2. RTP 穿越

解决了 SIP 穿越问题后，我们再来看 RTP 是如何穿越的。由于客户端的 SDP 信息中的 IP 地址是私网地址，因而 FreeSWITCH 无法直接给它发 RTP 包。而且，从 9.4.1 节我们也

⊖ 这里有一个实用的站长工具：http://tool.chinaz.com/Tools/URLEncode.aspx。

了解到，很多 NAT 设备都只有内网的主机曾经向外网主机发过包以后，才允许外面的包进入。因此 FreeSWITCH 使用了一个名为 RTP 自动调整的特性，即 FreeSWITCH 在 SIP 协商时给对方一个可用的公网 RTP 地址，然后等待客户端发送 RTP 包，一旦它收到 RTP 包以后，就可以根据对方发包的地址给它发 RTP 包了。当这种情况发生时，可以在 Log 中看到类似如下的信息：

```
[INFO] switch_rtp.c:4753 Auto Changing port from 192.168.1.124:50492 to 1.2.3.4:50492
```

虽然它不能解决所有问题，但总比不解决要好。当然，由于它没有约定使用 SIP 协商中指定的 IP 地址，这种解决办法可能有安全性问题，如黑客可以随机猜一些端口并向这些 RTP 端口发包，FreeSWITCH 收到后将远端地址调整到新的黑客的 IP 和端口，进而 RTP 数据全跑到黑客那里去了。所以 FreeSWITCH 为了防止这个问题，规定这种端口调整只能在电话开始的时候进行，一旦调整过，就不能再进行调整了。

这种自动调整是默认开启的，如果用户不需要该功能，可以在 Profile 中将其禁掉：

```
<param name="disable-rtp-auto-adjust" value="true"/>
```

或者只针对个别的呼叫来禁止自动调整，在呼叫时可以通过设置“rtp_auto_adjust=false”通道变量禁止。

3. 其他解决方案

我们前面讲到，FreeSWITCH 在默认的情况下就能很好地应对 NAT 的情况。但在某些网络环境下或对接某些客户端设备时（我们姑且称为“差”设备），以上手段可能还不够。FreeSWITCH 还提供了一些设置，这些设置默认是不开启的，如果开启了，可能这些“差”设备就可以工作了，但会影响“好”设备。

有些设备会自动进行 NAT 穿越，但却把 SIP 包里的参数改得乱七八糟。FreeSWITCH 提供了一个参数，其可以在 Profile 设置，以实现对这些 SIP 包进行“深度”检测，进而决定到底该用哪个 IP 地址。该参数是：

```
<param name="aggressive-nat-detection" value="true"/>
```

另外，FreeSWITCH 还有一族称为 NDLB[1]的配置参数，可以帮助某些“差”设备，如将 aggressive-nat-detection 参数配置到用户目录中，它会用接收到的 SIP 包的源地址改写 Contact 中的 IP 地址。

下面一个参数需要配置到 Profile 中：

```
<param name="NDLB-force-rport" value="true"/>
```

要理解 NDLB-force-rport 这个参数，先来说一下什么是 rport[2]。对于支持 rport 的设备，在发起请求时会在 Via 字段中带一个空的 rport 参数，如：

```
recv 813 bytes from udp/[1.2.3.4]:5060 at 04:05:37.660375:
```

---

[1] No Device Left Behind，不遗弃任何设备。参见：http://wiki.freeswitch.org/wiki/NDLB。

[2] 参见：http://www.ietf.org/rfc/rfc3581.txt。

```
----------------------------------------------------------------------
REGISTER sip:1.2.3.5 SIP/2.0
Via: SIP/2.0/UDP 192.168.0.2:5060;branch=z9hG4bK-d8754z-65d2be1ee849d851-1---
d8754z-;rport
Contact: <sip:1002@192.168.0.2:5060;rinstance=6622cfa07990b53c>
```

FreeSWITCH在收到该请求后，发现该SIP包的实际来源地址是1.2.3.4:5060，因此它会将响应包发往该地址，并在Via字段中设置rport的值为实际来源端口值，同时增加received参数，指定来源IP地址。它回应的SIP消息如下：

```
send 698 bytes to udp/[1.2.3.4]:5060 at 04:05:37.663393:
----------------------------------------------------------------------
SIP/2.0 200 OK
Via: SIP/2.0/UDP 192.168.0.2:56020;branch=z9hG4bK-d8754z-65d2be1ee849d851-1---
d8754z-;rport=5060;received=1.2.3.4
```

SIP客户端在收到该响应后，“学习”到了自己对应的外网地址，因而在接下来重发的注册信息中，Contact地址就可以填外网的地址了。

有意思的是，某些设备支持rport这一功能但在发送请求的时候在Via字段中又没有相应的rport参数。因而在FreeSWITCH开启NDLB-force-rport参数可以认为所有设备请求都带了rport参数，进而打开这些“差”设备的rport功能。需要指出的是，有些设备确定不支持rport，带了该参数可能会导致这些设备不能用，因而该参数的值也可以设置成safe（其他的取值还有client-only和server-only，读者可以自己尝试一下），即仅打开已知可用设备这一功能。

该参数是作用于整个Profile的。有时候在既要支持好设备又要支持差设备的情况下，是否使用该参数不能两全。解决的办法是可以再建立一个新的Profile，单独让这些差设备注册到新的Profile。

NDLB还有其他的参数，有的跟NAT相关，有的不相关。当然了，笔者希望大家永远都用不到这些参数，因为一旦涉及这些参数就会很麻烦，但如果有些设备必须要用这些参数，可以参考http://wiki.freeswitch.org/wiki/NDLB。

好了，接下来就要进入本节的正题了，让我们具体了解其他NAT穿越问题的解决方案。

（1）客户端解决方案

前面我们了解到，FreeSWITCH提供了多种手段来帮助客户端设备穿越NAT，并且在9.4.4节我们也提到，这些工作原来应该是在客户端做的。如果客户端能够知道自己将要被映射出去的外网IP地址和端口，那么它就可以直接在SIP/SDP消息中填上外网地址，因而FreeSWITCH就不用再费劲检测客户端是否有NAT了。

在9.4.4节我们讲到可以使用ifconfig.me之类的服务获取自己的外网IP，但SIP通信中还必须得知道映射的外网端口号。这一工作需要靠STUN[⊖]来实现。STUN的原理是：在公网上部署一台STUN服务器，位于NAT后面的客户端设备向它发送一系列的UDP包先在NAT设备上打一个“洞”，STUN服务器取到UDP包的来源地址以后，会回送相关的消息告诉该客户端它被映射的外网地址。

⊖ Session Traversal Utilities for NAT，即NAT会话穿越工具。参见：http://zh.wikipedia.org/zh-cn/STUN。

大多数的 SIP 话机及软电话客户端均可以选择是否启用 STUN 服务。使用 STUN 以后，SIP 消息的 Contact 头域中就可以直接填入外网地址，这时在 FreeSWITCH 端看到的注册信息如下（可以跟 FreeSWITCH 检测到 NAT 的情况对比一下）：

```
freeswitch> sofia status profile internal reg

Registrations:
=================================================================================
Call-ID:        MWI3OGU3OTk0ODc1ZTFhMmRhZWZmZmM1ZjkwOTVjMTM
User:           1002@115.28.33.221
Contact:        "1002" <sip:1002@119.40.26.181:29054;rinstance=98cf71bfe4dbb8bf>
Agent:          Bria 3 release 3.5.0b stamp 69410
Status:         Registered(UDP)(unknown) EXP(2013-08-17 12:54:14) EXPSECS(3646)
```

对比 9.4.1 节我们容易得出这样的结论，STUN 服务对于锥型 NAT 是有效的，但对于对称型 NAT 就无能为力了。这是因为锥型 NAT 对内网的同一个源地址和端口发往任何外网 IP 地址的数据都映射成同一外网 IP 和端口，而对称型 NAT 则对发往不同 IP 地址的包进行不同的映射，因而在对称 NAT 的情况下虽然客户端能从 STUN 服务器学习到外网 IP 地址和端口号，但是在客户端向真正的 SIP 服务器发 UDP 包时，在 NAT 设备上会映射出另外一个 IP 及端口号，因而先前从 STUN 服务器学习到的是无效的。

另外，考虑图 9-5c 所示拓扑结构，如果互相通信的两个 SIP 终端分别位于两个对称 NAT 之后，则它们是无法进行通信的，因为打通一个“洞”需要首先向对方发送数据，而在对称 NAT 的情况下谁都无法向先向对方发送数据，因而无法实现 UDP 通信。

TURN[⊖]技术主要用来解决对称型 NAT 的问题的，当然，它对于其他的 NAT 类型同样有效。使用它需要在公网上部署一台 TURN 服务器，该服务器作为一个“中间人”对双方的数据进行转发。FreeSWITCH 不支持 TURN[⊜]，在这里我们就不多讲了，有兴趣的读者可以参阅相关资料。

除了 STUN 和 TURN 外，还有 RSIP（Realm Specific IP）、symmetric RTP 等 NAT 穿越技术，但这些技术应用于不同的网络拓扑时都会各有利弊，无法在所有情况下都保证完美。所以人们一直在寻求一种足够灵活的方案，使用它可以在各种情况下对 NAT 穿透提供最优的解决方案。ICE 技术就是满足这一条件的解决方案。

ICE[⊜]的全称是交互式连通建立（Interactive Connectivity Establishment），它其实不是一个新的协议，而是综合利用现有的 STUN 和 TURN 等技术，使之在最合适的情况下工作，以弥补单独使用某种协议所带来的固有的缺陷。

在 SIP 通信中，ICE 通过扩展 SDP，为 RTP 媒体提供多个候选的地址（Candidate），这样两个 SIP 终端之间就可以尝试多个不同的候选地址找到一个最优的通信路径。比如，如果两个 SIP 终端处于相同的 NAT 后面，则它们可以直接用内网地址进行通信；如果位于不同的 NAT 后面，并且是锥型 NAT，则他们可以选择从 STUN 服务器获得的外网地址进行通

⊖ Traversal Using Relay NAT，即使用转发方式进行穿越的 NAT。参见：http://zh.wikipedia.org/wiki/TURN。

⊜ 但支持 TURN 的 SIP 客户端可以与 FreeSWITCH 配合工作。

⊜ 参见：http://zh.wikipedia.org/wiki/ 互动式连接建立。

信；如果是对称型 NAT，则只能通过 TURN 来进行通信了。

（2）FreeSWITCH 处于客户端的位置

以上我们讲的大部分是针对 FreeSWITCH 在公网上作为服务器端的情况。考虑图 9-5b 所示拓扑结构，当 FreeSWITCH 需要穿越 NAT 向外部的网关注册时，它就相当于处在客户端的位置[⊖]。我们一直在说客户端如果处于 NAT 之后，NAT 问题要自己解决，因此，FreeSWITCH 也有自己的解决方案。

首先，FreeSWITCH 支持通过 uPnP 或 NAT-PMP 协议[⊜]在路由器上"打洞"。打洞完成后它就知道了自己将要映射的外网地址了。当然，使用该协议需要路由器支持，在笔者的网络环境下，FreeSWITCH 启动时可以看到如下信息，证明 FreeSWITCH 确实检测到外网的地址了：

```
[INFO] switch_nat.c:420 Scanning for NAT
[DEBUG] switch_nat.c:170 Checking for PMP 1/5
[ERR] switch_nat.c:201 Error checking for PMP [general error]
[DEBUG] switch_nat.c:425 Checking for UPnP
[INFO] switch_nat.c:434 NAT detected type: upnp, ExtIP: '119.40.26.181'
[DEBUG] switch_nat.c:264 NAT thread configured
[DEBUG] switch_nat.c:275 NAT thread started
```

也可以在 FreeSWITCH 中使用如下命令进行验证：

```
freeswitch> show nat_map

port,proto,proto_num,sticky
5060,udp,0,0
5060,tcp,1,0
5080,udp,0,0
5080,tcp,1,0

4 total.
```

上述命令显示了 4 个端口映射关系，下列命令可以看到映射后的外网地址：

```
4 total. freeswitch> nat_map status

Nat Type: UPNP, ExtIP: 119.40.26.181
NAT port mapping enabled.
port,proto,proto_num,sticky
5060,udp,0,0
5060,tcp,1,0
5080,udp,0,0
5080,tcp,1,0
```

其中，ExtIP: 119.40.26.181 就表示 FreeSWITCH 是通过 uPnP 学习到的外网地址。

FreeSWITCH 检测到 NAT 以后，会设置 Profile 的外网 SIP 和 RTP 的 IP。下列命令显示了 external Profile 当前的 Ext-SIP-IP 和 Ext-RTP-IP：

⊖ 当然，其他的 SIP 客户端还可以向它注册，它还是服务器，如图 9-5c 所示。

⊜ uPnP 称为 Universal Plug and Play，即通用即插即用。NAT-PMP 称为 NAT Port Mapping Protocol，即 NAT 端口映射协议。它们主要用来在 NAT 设备上打洞，分别参见 http://zh.wikipedia.org/wiki/Upnp 和 http://en.wikipedia.org/wiki/NAT-PMP。

```
freeswitch@seven> sofia status profile external
=================================================================
Name                external
Domain Name         N/A
Auto-NAT            true
DBName              sofia_reg_external
Pres Hosts          192.168.1.124,192.168.1.124
Dialplan            XML
Context             public
Challenge Realm     auto_to
RTP-IP              192.168.1.124
Ext-RTP-IP          119.40.26.181
SIP-IP              192.168.1.124
Ext-SIP-IP          119.40.26.181
```

其中 Ext-SIP-IP 和 Ext-RTP-IP 是在与外网的服务器通信时用的。它在 FreeSWITCH 中默认的配置（在 external.xml 中）如下：

```
<param name="ext-rtp-ip" value="auto-nat"/>
<param name="ext-sip-ip" value="auto-nat"/>
```

其中，auto-nat 说明它试图自动检测到 NAT 外网的地址。如果路由器不支持 uPnP 及 NAT-PMP 协议，那么可以在这里填入一个 stun 服务器的地址，让它从 stun 服务器获取，如：

```
<param name="ext-sip-ip" value="stun:stun.freeswitch.org"/>
<param name="ext-rtp-ip" value="stun:stun.freeswitch.org"/>
```

注意，stun.freeswitch.org 可以做测试用，但它不保证永远可用。如果使用上述 STUN 服务不能正常获取 IP，则可以尝试其他 STUN 服务器。在网上可以找到好多免费的 STUN 服务器，当然需要自己运营 SIP 服务的应该自己搭建 STUN 服务器。

FreeSWITCH 提供了一个 stun API 命令用于测试 STUN 服务器是否能正确返回外网的 IP 及端口号。下列命令检测到服务器是正常的：

```
freeswitch> stun stun.freeswitch.org
119.180.70.163:50285
```

下列命令则不能返回正常的结果，因为我们使用的不是一个有效的 STUN 服务器：

```
freeswitch > stun www.freeswitch.org
-STUN Failed! [Timeout]
```

如果不使用 STUN，也可以手工找出路由器的外网 IP，有以下两种设置方法：

```
<param name="ext-sip-ip" value="autonat:119.40.26.181"/>
<param name="ext-rtp-ip" value="autonat:119.40.26.181"/>

<param name="ext-sip-ip" value="119.40.26.181"/>
<param name="ext-rtp-ip" value="119.40.26.181"/>
```

其中，上述两种配置方法是差不多的，只不过带了“autonat:”前缀以后 FreeSWITCH 会更智能一些，读者可以自己尝试，看看带与不带有什么区别，这里就不多讲了。最后，你也可以设置一个 DNS，如 host:sip.example.com，通过域名解析获得外网的 IP 地址。

（3）其他

一般来说，FreeSWITCH 足够智能，使用默认的配置就够用了。但 FreeSWITCH 并不能帮你解决所有 NAT 的问题，实际使用过程中还得具体问题具体分析。比如你可能为了兼容某些设备改了一些 Profile 的参数，同时导致另外一些本来好用的设备不好用了。或者，你修改了 ext-sip-ip/ext-rtp-ip 后外部的用户注册好用了，但 NAT 内部用户打电话却有问题了。大多数情况下，一个 Profile 通过设置适当的参数就能解决这些问题。但如果你遇到的不是上述的“大多数”的情况，那么可以通过创建多个 Profile 来解决，让某些设备注册到其中一个 Profile，另一拨人注册到另一个 Profile。当然同时使用多个 Profile 还需要一些设置技巧，我们将在后面的实战章节中深入探讨。

另外，考虑图 9-5c 所示拓扑结构，如果 FreeSWITCH 是企业内部的 PBX，员工可能经常会在办公室 A 或外部 B（家里）之间换来换去，那么它可能要在办公室里向内网地址 192.168.0.2 注册，在外部时就向 1.2.3.4 注册。每次改来改去比较麻烦。这种情况可以在企业内部部署 DNS 服务器通过 DNS 解决；另外有的路由器设备支持所谓“发夹”（Hairpin）的功能，通过这个功能可以解决该问题，有条件的读者也可以尝试一下，因为笔者没有见过这种路由设备，在这里就不多讲了。

9.5 小结

本章介绍了 mod_sofia 在 FreeSWITCH 中的位置及其基本概念。也讲到了其配置文件的结构，并对默认配置文件中的主要参数进行了解释。笔者深知，仅进行简单的解释还是不够的，实际的使用过程中可能会有好几个参数配合使用，也可能每一个参数都是一则故事，但是如果在本章把这些都展开，还是为时过早。一般在使用过程中仅需要用到少数的参数就够了，因此建议读者在这里先对这些参数有些直观的印象即可，后面的章节中我们会配合一些实例再来详细学习这些参数的用法。

此外，本章还花了很大的篇幅讨论 NAT 穿越问题。NAT 无处不在，所有人都可能遇到。但 FreeSWITCH 在默认情况下已经帮我们处理了大部分问题。如果你很不幸遇到了 NAT 的问题，那么也可以尝试修改本章提到的参数看能否解决。对于更复杂的 NAT 问题，解决起往往需要花很多时间和精力，有时候并不能仅靠修改几个参数实现，可能还需要改变网络的拓扑结构、调整 SIP 终端设备或路由器的参数等[⊖]。因此，对于初学者而言，还是建议在局域网上学习，等熟悉了各种配置参数以及掌握了一定的调试技术以后再尝试解决 NAT 穿越问题。解决 NAT 穿越问题时比较有效的方法是进行抓包分析，有时候可能需要在多个点（如话机侧、路由器上、服务器）上抓包比较。然后，尝试修改本章提到的各种参数，看有什么不同。总之，NAT 问题复杂归复杂，有了这些基础知识再加上一定的经验，多花点时间，相信所有人都能解决。

⊖ 有一次，在解决 NAT 引起的单通问题时我们尝试了各种手段就是收不到 RTP，后来禁用了路由器的“启用快捷转发”功能才把问题解决。能用这样的方法解决问题，多半是靠运气。

第二部分

实 战 篇

在前面的章节中，我们介绍了与 SIP 以及 FreeSWITCH 相关的一些基础知识。所谓“理论与实践相结合”，掌握任何一门技能都少不了不断的练习。本篇收集了笔者在多年的工作过程中的一些实例，以及一些博客文章。有一些是笔者在学习的过程中留下来的，有一些则是在实际的环境中应用的。无论如何，笔者希望通过这些实际的例子，抛砖引玉，与大家共同学习和提高。

读完本篇，你可以成为一名高级 FreeSWITCH 维护工程师。

第 10 章
基本技能

通过前面的章节的学习，我们了解了 FreeSWITCH 必备的一些背景资料和基础知识，并掌握了 FreeSWITCH 中的基本概念和基本理论。有了这些概念和理论为基础，我们接下来就可以着手进行实际练习了。

在本章，我们先来学习一些基本的技能，为以后的实践打下一个好的基础。

10.1 调试与排错

在实际的应用场景中，我们不可避免地会遇到各种各样的问题，而要解决这些问题一般需要根据问题的现象、系统提供的日志、线索等查找原因。因此，在进一步练习之前，我们先来学习一下解决问题的思路、方法和技巧。熟练掌握了这些方法和技艺，以后无论遇到什么样的问题就都会很容易解决了。

解决问题首先要找到相关的线索，并加以分析。在前面的章节中我们也穿插着介绍过一些调试的手段，在本节我们再系统讲解一下。

10.1.1 解决问题的一般方法和流程

解决问题一般要经过以下几个步骤：发现问题、定位问题、分析问题、解决问题。下面我们在对这些步骤进行讲解时并不上纲上线，而是以一个例子来简单地进行说明，目的就是教会大家如何发现并准确地定位问题。当定位了问题故障点后，就比较容易分析和解决该问题了。

假设我们遇到这样一个问题：Alice 向我们报告说，她通过 FreeSWITCH 呼叫 Bob，能呼通，但没有声音。

这确实是一个问题，因为电话的基本功能就是为了双方能互相听到对方的声音。不过，在这里我们虽然收到了这个问题报告，但对该问题的实际情况还了解得不够清楚。比方说，首先，也是最基本的，是两端都听不到声音，还是只有一方没有声音？其次，是每次都听不

到，还是偶尔听不到？如果是偶尔，在什么情况下听不到？再次，Alice 和 Bob 使用的是什么 UA，软电话还是硬件的 SIP 话机？都是什么型号的？还有，他们所在的网络情况是什么样的？他们是用 ADSL、小区宽带，还是用手机上网？……

一般来说，Alice 只是个普通的电话用户，她可能不知道这些技术细节。但在这种情况下，我们就需要多问一下 Alice，以确认这些问题实际上否与 Alice 描述的一致，以及能否重现等。了解了这些信息，我们才能进行下一步的定位和分析。总之，在这里，有两个问题是很值得考虑的：

- 核实问题的现象。即假定 Alice 是不可信的，就算她没说谎，但有时问题的实际现象 Alice 可能描述不清楚，或者可能跟她描述的完全不一样。如果有条件最好亲自试验一下。
- 问题能否重现。大家解决问题多了就会知道，如果一个问题总是能重现，那么一般是比较容易解决的。最难解决的问题就是不能重现的问题，或者不知道什么时候能重现的问题。说白了，就是电话有时候好，有时候不好，但至于什么时候好，什么时候不好，说不上来，总之就是没有任何规律。

在发现问题之后，就要尝试定位问题，即找出该问题出在什么地方。要想定位问题，要先有一个思路。笔者一般建议使用“分段法”和“换位思考法”来定位问题。所谓分段法，就是说，一个问题可能涉及很多环节，我们首先要把这些环节从逻辑上和功能上分开，一段一段地检查。所谓换位思考法，结合我们本节的例子可以理解为，如果 Alice 打给 Bob 没有声音，那么，Bob 打给 Alice 有没有声音？能不能把 Alice 和 Bob 的话机对调一下再试一次？或者，有条件的话，给 Alice 或 Bob 换其他的话机试试。

这种定位问题的方法在对整个系统不了解的情况下，可以首先确定是不是话机终端的问题。使用这种方法不需要很高级的技术，但它却很有效，因为话机终端出现问题的情况会比较常见。比如，如果一方用的是软电话，还可以考虑是否是声卡有问题。好多人也会将喇叭静音，甚至出现音箱没开电源或耳机没插到正确的插孔里的情况。

当上面这些方法找不到具体原因时，就考虑分段法。如，首先让 Alice 拨打默认的 9196，看能否听到自己的声音。到现在我们都很熟悉了，9196 是默认的回音测试（echo）功能，如果 Alice 打 9196 都听不到声音，就先不要管 Bob 了，因为呼叫还没到达 Bob，到目前为止还跟他一点关系都没有呢，所以要先集中精力解决 Alice 打 9196 的问题再说。

当然 Alice 打 9196 听不到声音的原因可能有很多，我们还需要一步一步地查。在这里首先建议关掉防火墙——Linux 上的 iptables、Windows 防火墙等。对新手而言，这些太容易导致问题了。因此，简单地关掉，会省掉很多麻烦[⊖]。

确认防火墙没有问题后，可以通过抓包看有没有 RTP 流到达或从服务器发出。我们将在下一节讲解使用 Wireshark 抓包的知识。在本节，我们来学习一个 uuid_debug_media API 命令。这是一个很有用但常常被忽略的命令。首先，在 FreeSWITCH 控制台上可以使用

⊖ 注意，如果你在病毒很多或者很容易受攻击的电脑或服务器上学习 FreeSWITCH 而又不明白关掉防火墙的风险的话，建议不要这样做。如何部署防火墙已超出了本书的范围，故这里不过多介绍了。

show channels 命令来找到当前通话的 Channel 的 UUID，然后使用 uuid_debug_media 调试媒体相关的信息。下面是笔者调试 echo 的情况（排版需要，截断了一些无关紧要的过长的行）。

首先找到当前通话的 UUID，下列命令的结果中显示的第一个字段就是 UUID：

```
freeswitch> show channels

uuid,direction,created,created_epoch,name,state,cid_name,cid_num,ip_addr,......
b4fae306-3f78-4d91-bcde-a2e95b0f9c1d,inbound,2013-07-23 12:04:38,1374552278,......
```

然后，使用 uuid_debug_media 命令查看调试输出。其中，该命令的第一个参数是 Channel 的 UUID，接下来是欲调试的方向。方向有 read 和 write 两种（即读和写，也即收和发，都是相对于 FreeSWITCH 而言的，下同），也可以使用 both 参数表示双向都调试。最后一个参数是 on 或 off，分别表示打开或关闭调试。命令的输出如下：

```
freeswitch> uuid_debug_media b4fae306-3f78-4d91-bcde-a2e95b0f9c1d both on

+OK Success
R sofia/internal/1002@192.168.7.5 b=172 192.168.7.5:17318 192.168.7.5:50050
pt=0 ts=3717002357 m=0
W sofia/internal/1002@192.168.7.5 b=172 192.168.7.5:17318 192.168.7.5:50050
pt=0 ts=3717002357 m=0
R sofia/internal/1002@192.168.7.5 b=172 192.168.7.5:17318 192.168.7.5:50050
pt=0 ts=3717002517 m=0
W sofia/internal/1002@192.168.7.5 b=172 192.168.7.5:17318 192.168.7.5:50050
pt=0 ts=3717002517 m=0
```

R 与 W 分别表示 Read 和 Write，即收和发。如果能看到 R 和 W 了，说明已正确收发媒体流。如果还是听不到声音，那可能需要更深入地检查了。更深入的检查在这里我们就不过多地展开了。我们在此再简单说一下上面调试输出的意义。b=172 表示收、发 172 字节，它等于 160 字节的 PCMU 语音净荷（20ms）加上 12 个字节的 RTP 包头。17138 和 50050 分别为本地和远端端口号。pt=0 说明 Payload Type（载荷类型）为 0，即 PCMU 编码的语音数据。ts 为时间戳，可以看出它在收和发方向上分别是以 160 递增的。最后 m 为 RTP 的 Marker 位，一般是 0。

注意，上述我们使用了 echo 进行测试，它一般用于简单测试双向语音的情况，即 FreeSWITCH 作为一个“回音壁”，会等待客户端将 RTP 包发过来然后原样返回。如果由于防火墙或某种原因导致 FreeSWITCH 收不到 RTP 包，则它也就没有数据可发了。

在某些情况下，会只有单向的语音流，这种情况称为单通。测试单通的情况，可以用上面的 uuid_debug_media 对“收”和“发”分别测试。在没法执行上述命令的情况下，也可以使用更直观的方法进行测试。例如，出于简单起见，我们先测试“发”。拨打默认的 9664，FreeSWITCH 就会播放保持音乐，如果对方听不到的话，就需要在对方的电脑或话机上抓包看 RTP 包是否到达。如果没有到达，说明网络有问题，如果到达了还没有声音，那就可能是对方终端的问题，或扬声器或耳机的问题。

然后，倒过来测试是否能“收”到 RTP 流。先建一个 Dialplan 路由到 record App 进行录音，录音完成后听一下录音文件是否完整即可判断是否正确收到 RTP 流。录音的

Dialplan 设置如下：

```
<action application="record" data="/tmp/test.wav"/>
```

或者直接使用如下回拨命令进行录音：

```
freeswitch> originate user/1002 &record(/tmp/test.wav)
```

如果一切正常，说明 Alice 端没有问题了。如果有问题我们也可以使用上面的步骤定位到问题并进行解决。总之，当我们解决完了 Alice 端的问题后，再用同样的方法解决 Bob 的问题。保证两端都没有问题后，再尝试直接从 Alice 呼叫 Bob，看 bridge 后是否还有其他问题。一步一步，直到问题解决。

最后需要说明，没有媒体流或者媒体流不正常也可能是收到 SIP 消息错误引起的，我们后面还会讲到跟踪 SIP 消息的方法。

10.1.2 查看日志

在有些情况下，呼叫可能直接不通，即无法正常建立呼叫。这时候我们就需要查看系统的日志。

在日志中，"警告"（WARNING）和"错误"（ERROR）是级别比较高的日志，"调试"（DEBUG）是级别比较低的日志，但其能显示更多的细节。在 FreeSWITCH 控制台上，可以通过 console loglevel debug 命令打开 DEBUG 级别的日志。

我们还是以 Alice 呼叫 Bob 为例，如果 Alice 发起呼叫后什么日志也不显示，那么肯定呼叫没有到达 FreeSWITCH，此时应检查客户端是否配错了 IP 地址或防火墙是否有问题。如果呼叫到达了，还是出错，那么参照第 6 章中的介绍，查看是不是 Dialplan 设置的问题（还记得"从绿色的行开始看"吗？）。另外，日志中一般有挂机原因（Hangup Cause），比方说 CALL_REJECTED 一般代表呼叫被拒绝，可能是认证的用户名或密码不对。如果 Alice 方面没什么问题，错误也可能是 Bob 那端引起的，比如 Hangup Casuse 为 USER_NOT_REGISTERED 就说明 Bob 没有注册，因而呼叫无法继续进行。

有时候，呼叫 Bob 可能需要经过一个网关出去，这时候也可以采取"分段"的调试方法，如使用类似如下的命令看看呼叫的后半段有没有问题（即从 FreeSWITCH 中直接呼叫 Bob 有没有问题）：

```
freeswitch> originate sofia/gateway/gw1/Bob &echo
```

其中，gw1 为呼叫 Bob 使用的网关的名字。上述命令是阻塞的，因而在 Bob 对应的网关没有任何反应的时候（不回任何 SIP 消息），可能会阻塞较长时间。当 FreeSWITCH 控制台被阻塞时，将不能输入任何命令。如果要解决这个问题，可以打开另一个终端，用 fs_cli 连接 FreeSWITCH，然后使用 show channels 找到对应的 Channel，并且用 uuid_kill <channel UUID> 将该 Channel 释放。当然，为了避免产生这种阻塞的情况，可以提前使用 bgapi，如：

```
freeswitch> bgapi originate sofia/gateway/gw1/Bob &echo
```

bgapi 可以使 originate 在后台（新的线程中）执行，因而不会阻塞控制台。

另外，在社区中（邮件列表或QQ群）中提问时，帮助你的人通常也会要求提供DEBUG级别的日志，以便于进行分析。

除了看日志以外，FreeSWITCH也支持现场抓包，使用下列命令可以在FreeSWITCH控制台上实时看到SIP消息，如：

```
freeswitch> sofia profile internal siptrace on
freeswitch> sofia profile external siptrace on
```

上面的命令可以开启不同Profile的SIP抓包。当a-leg和b-leg分别在不同的Profile上时可以分别来看，避免消息太多造成干扰。当然，初学者可能不知道实际的SIP消息应该走哪个Profile，这时可以使用以下命令打开所有Profile的抓包：

```
freeswitch> sofia global siptrace on
```

当然，也可以随时关闭抓包：

```
freeswitch> sofia global siptrace off
```

有时候，还能遇到更奇怪的情况，在试上了述各种办法后还是找不到原因。如果怀疑是SIP协议栈底层的问题（那么可能是SIP兼容性的问题或FreeSWITCH本身的BUG），就要打开底层协议栈的调试信息。开启Sofia协议栈底层的调试器的方法是使用以下命令：

```
freeswitch> sofia loglevel all 9
```

可以使用以下命令关闭所有底层的调试：

```
freeswitch> sofia loglevel all 0
```

Sofia层的日志信息会有很多，如果你比较熟悉的话（一般是开发者），也可以仅打开某一类的日志，如：

```
freeswitch> sofia loglevel nua 9
```

具体可以参考sofia命令的帮助信息，如直接输入sofia命令不带任何参数会显示一个帮助，里面可以看到下面一行：

```
sofia loglevel <all|default|tport|iptsec|nea|nta|nth_client|nth_server|nua|soa|
sresolv|stun> [0-9]
```

上述帮助信息中可选的参数表示，可以选择调试Sofia协议栈的各个部分。协议栈底层的知识超出了本书的范围，我们在此就不多介绍了。

小技巧

有时候，尤其开启了测试以及SIP Trace后可能会收到大量的消息，不停滚动屏幕会令人很难定位到有问题的消息。笔者在调试时遇到这类问题时一般会使用fs_cli连接到FreeSWITCH，开启相应的DEBUG级别和跟踪，打个测试电话，然后使用Ctrl+D快捷键或/exit命令退出，这样就可以避免屏幕滚动了。

在实际应用中，遇到的问题可能还有很多。不要慌，冷静地思考和分析一下，一般很容易定位到问题。即使自己不能准确地定位问题，在寻找帮助时，把你通过上述步骤了解到的

情况跟有经验的人说一下，一般他们就能很快帮你定位到问题。最忌讳的就是什么也不做，直接上 QQ 群里发问："我装了 FreeSWITCH，怎么打不通电话啊？"那样的话，问了等于白问，谁都帮不了你。

10.2 使用外部工具抓包

虽然在 FreeSWITCH 内部也可以抓包，但有时候需要对 IP 包进行更全面的分析，或者存档，因此经常需要使用外部的工具抓包。

现有的大部分工具都是基于 libpcap 实现的，它们在不同的场景下各有优势，这里我们来简单介绍一下[⊖]。由于抓包是一种非常底层的操作，因此在大多数平台上通常需要用 root 或管理员用户来执行。他们大部分也都支持将抓包结果存储在本地 PCAP 格式的文件中，这种格式的文件后续可以使用其他工具（如 rtpplay 和 Wireshark 等）进行重放和分析。

10.2.1 tcpdump

tcpdump[⊜]是经典的抓包工具。比如下面的命令可以仅抓取 5060 端口上的 SIP 包：

```
# tcpdump -nq -s 0 -A -vvv -i eth0 port 5060
```

其中，-n、-q 表示不进行域名翻译及减少输出内容；-s 0 表示不限制包长，即争取抓最大的长度；-A 表示以 ASCII 方式输出，这样用眼看起来比较直观；-v 表示显示的详细程度，"v"越多则越详细；-i 表示使用指定的网卡；port 5060 表示过滤器，这里我们只关心 5060 端口上的 SIP 包。

另外，可以使用 -w 将结果写入文件中，如下列命令将结果写入 /tmp/dump.pcap 文件中：

```
# tcpdump -nq -s 0 -i eth0 -w /tmp/dump.pcap port 5060
```

当还需要分析 RTP 流时，可以将 port 5060 简单改成 udp，这样就可以抓取所有的 UDP 包：

```
# tcpdump -nq -s 0 -i eth0 -w /tmp/dump.pcap udp
```

有时候，如果你是在服务器上抓包，并想只抓某个用户的包，可以根据用户的 IP 地址来进行过滤。当然这需要先找到用户实际的 IP 地址，通过如下命令可以找到用户 1002 的 Contact 地址：

```
freeswitchlocalhost> sofia status profile internal reg 1002

Registrations:
=================================================================================
Call-ID:          ZjUwMGY3ODM5M2E2ZGM0MWFiZGRiNDhkNThkZjk3NzQ
```

⊖ 我们在以后的章节中更专注于具体应用，因而各种工具的具体安装步骤在此就不多费笔墨了。一般来说，在 Linux 系统上，都是 yum install（CentOS/Redhat）或 apt-get install（Ubuntu/Debian）后面加上软件包的名字即可。其他情况也读者也可以根据软件包的名字自行搜索相关的安装资料。

⊜ 参见 http://www.tcpdump.org/。

```
User:          1002@192.168.7.5
Contact:       "Seven" <sip:1002@192.168.7.5:65272;rinstance=2d4d57704e73155d>
Agent:         Bria 3 release 3.5.0b stamp 69410
Status:        Registered(UDP)(unknown) EXP(2013-07-23 15:17:51) EXPSECS(616)
Host:          seven.local
IP:            192.168.7.5
Port:          65272
Auth-User:     1002
Auth-Realm:    192.168.7.5
MWI-Account:   1002@192.168.7.5
```

然后就可以只针对该用户抓包了（从 Contact 地址中我们知道它的 IP 和端口号分别是 192.168.7.5 和 65272）：

```
tcpdump -i eth0 -s 0 -A host 192.168.7.5 and port 65272
```

有时候，我们也需要抓取所有的 RTP 包进行分析。在 FreeSWITCH 中，默认的端口号范围是 16384 ~ 32768，因此我们可以使用如下命令抓包：

```
tcpdump -i eth0 -w /tmp/sip-rtp.pcap "udp and (port 5060 or port 5080 or
portrange 16384-32768)"
```

上面的命令中，使用 or（或）定义了多个过滤器参数，它分别代表抓 5060、5080 以及端口范围在 16384 至 32768 之间的包。

更详细的说明参见 http://wiki.freeswitch.org/wiki/Packet_Capture。

10.2.2 tshark

tshark[⊖]是 Wireshark 的命令行版。使用方法与 tcpdump 类似，如可以直接在命令行上运行 tshark。下面是笔者电脑上的输出结果：

```
# tshark
Capturing on en0
  0.000000 192.168.1.119 -> 123.125.115.179 TCP 54 61729 > http [FIN, ACK]
Seq=1 Ack=1 Win=16384 Len=0
  0.153641 173.194.79.109 -> 192.168.1.119 TLSv1 164 Application Data
  0.153644 173.194.79.109 -> 192.168.1.119 TLSv1 247 Application Data
  0.153645 173.194.79.109 -> 192.168.1.119 TLSv1 130 Application Data
  0.153672 173.194.79.109 -> 192.168.1.119 TLSv1 568 Application Data
```

其中，Capturing on en0 表示从第一块网卡上抓包（注意，笔者用的是 Mac 系统，Mac 上的 en0 相当于 Linux 上的 eth0）。它等价于以下命令：

```
# tshark -i en0
```

另外，也可以将抓包结果写入 pcap 文件中，如在 Linux 上可以这样用：

```
# tshark -i eth0 -w /tmp/dump.pcap
```

与 tcpdump 比较起来，tshark 能“认识”更多的协议，比如下面的命令可以很直观地看到收发的 SIP 包：

⊖ 参见 http://www.wireshark.org。

```
# tshark host 192.168.1.9 and port 5060
Capturing on en0
  0.000000 192.168.1.119 -> 192.168.1.9
  SIP 571 Request: REGISTER sip:192.168.1.9    (fetch bindings)
  0.010322  192.168.1.9 -> 192.168.1.119
  SIP 709 Status: 401 Unauthorized    (0 bindings)
  0.010972 192.168.1.119 -> 192.168.1.9
  SIP 817 Request: REGISTER sip:192.168.1.9    (fetch bindings)
  0.030214  192.168.1.9 -> 192.168.1.119
  SIP 587 Status: 403 Forbidden    (0 bindings)
```

上述的 SIP 包是笔者在调试的时候在本机上抓到的，笔者在注册一个远程服务器时由于密码错误，认证失败，因而收到了 403 Forbidden 消息。

当然，tshark 还有很多其他的命令行参数，我们在这里就不多介绍了。我们将在 10.3.1 节讲解它对应的图形界面版本的用法。

10.2.3 ngrep

ngrep[⊖]也是一个非常好用的抓包工具（类似于经典的 UNIX 命令行工具 grep），它可以提供类似于 FreeSWITCH 内部的抓包输出的显示方式，显示比较直观。笔者常用的选项组合是：

```
# ngrep -p -q -W byline port 5060
```

其中，-p 表示不使用混杂模式[⊜]；-q 表示使用安静模式，仅输出包头和相关包的载荷（如果有的话）；-W 表示选择一种显示输出方式，对于基本文本的协议来说，使用 byline 可以按换行符进行换行，显示比较直观；最后的 port 5060 是一个过滤器，即我们只想看我们关心的包。下面是与上一节最后一个例子对应的 ngrep 抓包的结果：

```
# ngrep -p -q -W byline host 192.168.1.9 and port 5060
interface: en0 (192.168.1.0/255.255.255.0)
filter: (ip) and ( host 192.168.1.9 and port 5060 )

U 192.168.1.119:3454 -> 192.168.1.9:5060
REGISTER sip:192.168.1.9 SIP/2.0.
Via: SIP/2.0/UDP 192.168.1.119:3454;branch=z9hG4bK-d8754z-4d3eaa270f669f7b-1---d8754z-;rport.
Max-Forwards: 70.
Contact: <sip:607@192.168.1.119:3454;rinstance=9066826f34316cd5>.
To: "607"<sip:607@192.168.1.9>.
From: "607"<sip:607@192.168.1.9>;tag=42c9367f.
Call-ID: NTcwNDJhOWE3MDE0OTAwMmQzZDFkNjdlYTZjOGVhMDk.
CSeq: 1 REGISTER.
Expires: 3600.
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO.
User-Agent: Bria 3 release 3.5.0b stamp 69410.
Content-Length: 0.
.
```

⊖ 参见 http://ngrep.sourceforge.net/ 。

⊜ 混杂模式即 Promiscuous Mode，如果将网卡设置成混杂模式便可以在一台主机上接收局域网上所有主机的包（当然这也取决于 Hub 或交换机的设置），也就是说有可能抓到发往其他主机的包。

```
U 192.168.1.9:5060 -> 192.168.1.119:3454
SIP/2.0 401 Unauthorized.

U 192.168.1.119:3454 -> 192.168.1.9:5060
REGISTER sip:192.168.1.9 SIP/2.0.

U 192.168.1.9:5060 -> 192.168.1.119:3454
SIP/2.0 403 Forbidden.
```

注意

上面的抓包结果中，为了节省篇幅，我们仅保留了第一个完整的SIP消息，其他的仅保留了起始行。

从上面的抓包结果中可以很清楚地看出数据包的传输方向以及SIP协议的具体内容。

除了host、port这样的过滤器外，ngrep还支持针对内容的过滤。这在比较繁忙的服务器上（比如有大量SIP包的情况下）比较有用。下面的命令可以过滤仅含有139xxxxxxxx(可能是个手机号码）字符串的内容：

```
# ngrep -p -q '139xxxxxxxx' -W byline port 5060
```

具体的过滤后的结果输出与上面讲到的类似，在这里我们就不多讲了。更多的选项可以查看ngrep的帮助信息。

10.2.4 pcapsipdump

与上面的抓包工具不同，pcapsipdump[⊖]有一个很好的特性，它能将不同通话IP包存到不同的文件里，其中某一路通话所有的SIP和RTP数据都存到同一个文件中。这样在同时有大量通话的时候抓包分析比较有用。使用方法如下：

```
pcapsipdump -i eth0 -d /tmp/sipdump/
```

上述命令将抓到的包文件都以PCAP格式存放到/tmp/sipdump这个目录中。后续可以使用其他工具针对个别的呼叫进行分析。这在服务器话务量比较高的时候比较有用。

pcapsipdump工具也有其他参数，在这里我们就不多介绍了。建议读者把这里介绍的各种工具都试一下，这样在遇到问题时就可以比较熟练地使用合适的工具查找并解决问题了。

总之，本节介绍的4种抓包工具中，前两种除了语法稍有不同外，在功能上是差不多的，它们都能抓包并存为PCAP格式的文件，供以后分析。而ngrep主要的特点是类似于UNIX grep，在文本界面下实时看起来比较方便。最后的pcapsipdump是专门针对SIP的抓包工具，在抓包的同时能将不同的通话分别存放到不同文件中。在使用时，可根据需要选用合适的工具。

10.3 使用Wireshark抓包并分析呼叫

在涉及FreeSWITCH与其他系统的对接时，由于大家对SIP信令及相关RFC的理解有

⊖ 参见：http://sourceforge.net/projects/pcapsipdump/。

偏差，或者具体实现的取舍不一样，就可能造成信令间细微的差别。虽然在 FreeSWITCH 内部也可以抓包，但如果系统中有大量的呼叫，使用 FreeSWITCH 抓包就有些“看不过来了”。这时候，就要使用我们上一节讲到的各种抓包工具。

本节将介绍一款专用的抓包工具——Wireshark，其附带了更多的分析工具，可以比较全面地分析呼叫。

如果 Wireshark 跟你的 FreeSWITCH 在同一台机器上，可以直接使用 Wireshark 抓包；如果你想抓远程 Linux 服务器上的包，而一般远程机器没有或不打开图形界面，则可以在 Linux 上安装 tcpdmp 或 Wireshark 的命令行版本 tshark。

10.3.1　使用 Wireshark 抓包

我们上一节讲到的各种抓包工具都是命令行版本的，比较适合应用在没有图形界面的服务器上。在这一节，我们要讲的是一个图形界面的抓包工具 Wireshark，也就是 tshark 的图形界面版。当然，除了它有一个比较好的图形界面外，它更强大的功能是能识别主流的各种通信协议，因而可以很方便地分析协议的内容（它甚至专门有一个 Telephony 主菜单，专门用于分析各种电话类协议）。

Wireshark 是跨平台的，在主流的平台上都有相关的安装包。关于如何下载和安装 Wireshark，在这里我们就不多介绍了。下面我们来看一些实际抓包分析的例子。

在进行抓包前，首先打开 Wireshark，按最左边的第一个图标选择一个网卡设备，如图 10-1 所示。

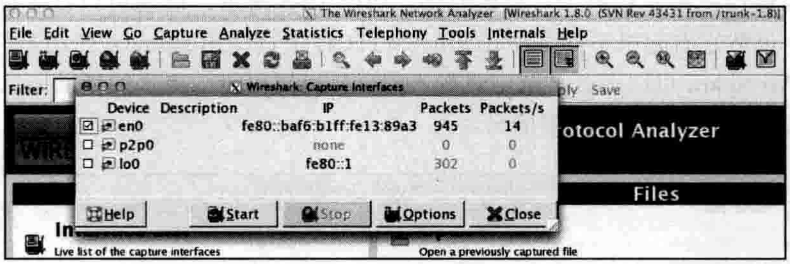


图 10-1　选择一个网卡设备进行抓包

在笔者的 Mac 电脑上，网卡的名字是 en0，它相当于 Linux 上的 eth0（在 Windows 上可能会有其他的名字，名字不确定）。一般来说，在选择网卡的界面上稍微停留一会，相关网卡后面会有带数字的流量指示，应该能比较容易找到对应的网卡。

选择一个网卡后，就可以单击 Start 开始抓包了。这时候，笔者打开了一个视频电话（默认的 9196 号码），此时可以看到 SIP 包以及许多 RTP 的包[⊖]。

⊖ 注意，笔者的 FreeSWITCH 与 Wireshark 在同一台机器上，而使用的软电话在另外一部手机上。如果所有设备在同一台机器上则抓不到包（当然有别的办法解决），因为要保证数据包是经过网卡的。如果读者想抓取与远程 FreeSWITCH 的通话，可以在与 Wireshark 相同的机器上装软电话，这样也能抓到包，只不过是客户端这一侧的。

我们这时候就可以挂断电话了。电话挂断后，我们可以停止抓包，否则数据量大了以后看起来就比较麻烦了。另外，为了方便以后分析，我们也可以将抓到的数据包保存起来，一般保存为 PCAP 文件，这样的文件 Wireshark 能自动打开。

10.3.2 使用 Wireshark 对抓包进行分析

Wireshark 可以分析使用 tcpdump 或 pcapsipdump 抓下来的 pcap 包。在这里，我们直接分析上一节抓到的包。

1. 分析 SIP 包

首先，我们先来看 SIP 信令。由于我们抓到了很多包，为了看起来方便，可以使用 sip 过滤器，只看我们关心的包。如图 10-2 所示，我们可以在 “Filter:” 一栏中输入 “sip”，便可以只显示所有的 SIP 包。

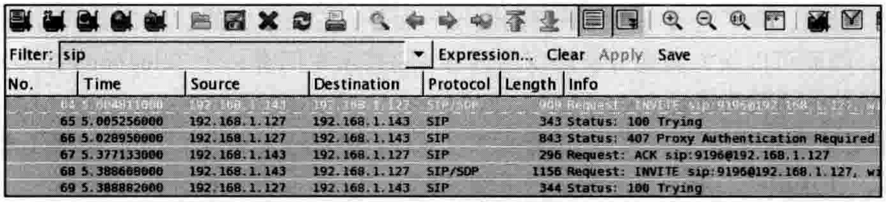


图 10-2 使用了 sip 过滤器只显示 SIP 包

我们可以通过鼠标定位到 INVITE 消息，如图 10-3 所示，读者可以在下面的框里看到消息的内容。虽然它看起来不如直接在 FreeSWITCH 中直接用 siptrace 抓出来的纯文本的包直观，但是 Wireshark 可以对 SIP 消息做深入的分析，可以显示更多的帮助信息。图 10-3 显示了 SIP 中的 INVITE 消息的详细情况。

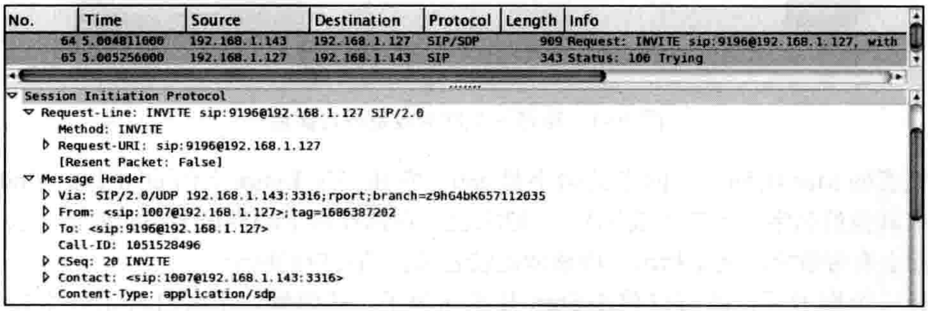


图 10-3 在 Wireshark 中查看 INVITE 消息

Wireshark 甚至有专门分析 VoIP 通话的工具。在主菜单中，选择 Telephony → VoIP Calls 可以看到所有的 VoIP 呼叫，如图 10-4 所示。

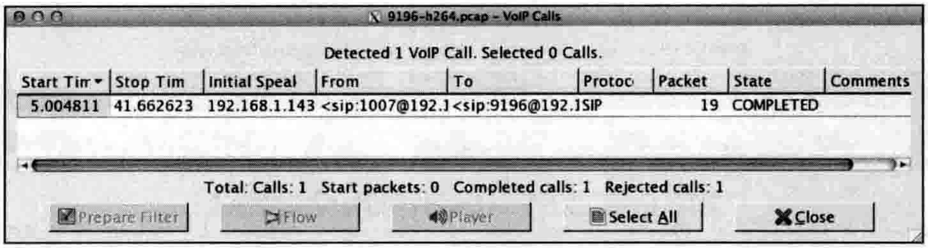


图 10-4 一路 VoIP 通话

此时，选择一路呼叫并单击 Flow 按钮，就可以看到详细的 SIP 呼叫流程，如图 10-5 所示。

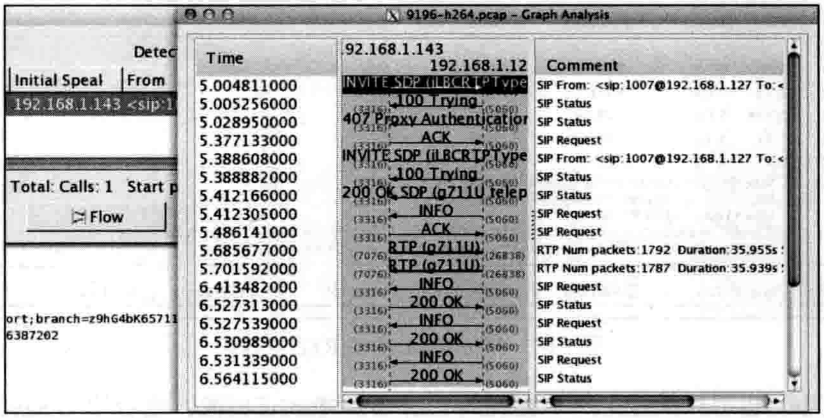


图 10-5 Wireshark 中分析呼叫流程

有了前面的 SIP 基础，读者就可以按照这个流程自己试一下了。打几个正常的或者不正常的电话，比较一下它们的异同。

另外，在上一步中，Flow 按钮的旁边有一个 Play 按钮，单击这个按钮以后再单击 Decode 按钮可以将音视频流的数据进行解码，并可以播放音频流里的声音。注意，如果音频不是以 PCMU 或 PCMA 格式编码的，则不能播放。关于 RTP 流的详细分析我们将在下一节进行讨论。

2. 分析 RTP 包

在 Wireshark 中也可以分析 RTP 包。在 Filter 栏中输入“RTP”就可以看到所有的 RTP 包。一般来说，可以先肉眼大体看一下收发是否规律。详细的分析可以使用 Telephony → RTP → Show All Streams 显示所有的 RTP 流，如图 10-6 所示，它显示了我们这次呼叫中涉及的两路 PCMU 的音频流和两路 H264 的视频流。

先选择一路音频流，通过单击 Find Reverse 按钮，可以找到并选择反向的流。然后单击 Analysis 就可以看到两路 RTP 流的详细分析，如图 10-7 所示。

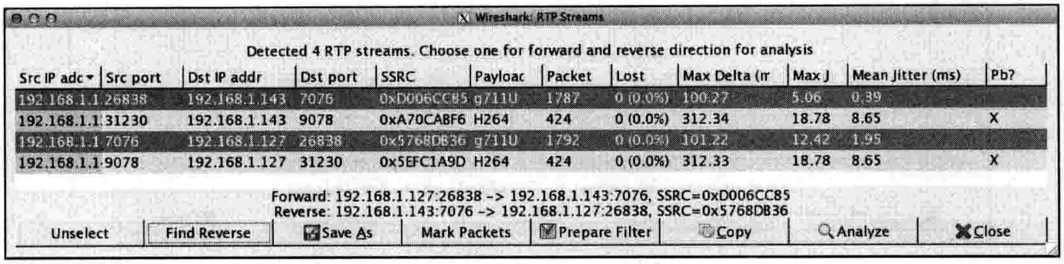


图 10-6　一个呼叫中的 4 路 RTP 流

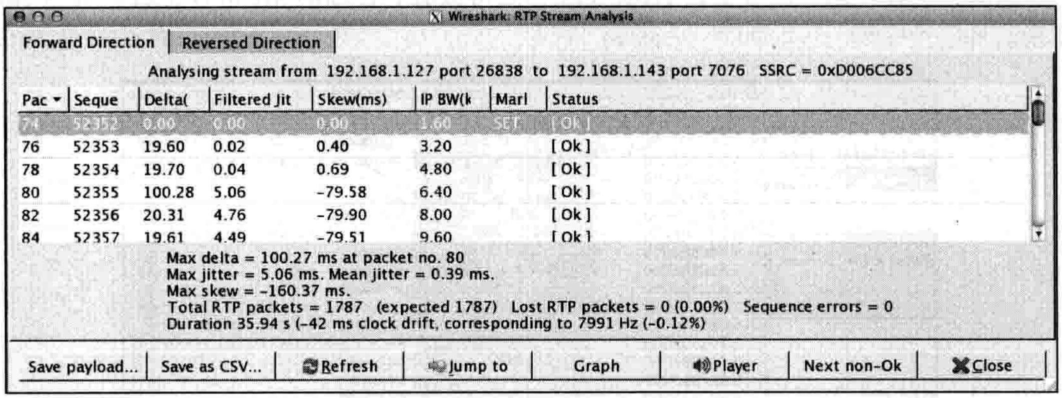


图 10-7　Wireshark 中的 RTP 流分析

在分析中，也可以通过上一节提到的 Play 按钮播放音频流，或者使用 Save Payload 按钮将音频数据存到单独的文件中。如果使用 AU 格式，则可以在同一个文件中保存双向的音频流；但如果使用 RAW 格式，只能将单向的音频流分别存到不同的文件里。另外，如前面所说，如果语音编码不是 PCM 格式的话（如 G729），则音频流只能存成 RAW 格式，然后再用其他工具去分析。

除此之外，还可以使用 Graph 按钮生成某一路流相关的统计图，如果图表显示比较规律，那一般说明是比较好的，如果生成的图表不是很规律，那就说明可能是网络上有丢包、延迟或抖动等。(我们曾经在第 9 章讲过这几个指标，这里就不多讲了)。解决音频或视频质量的问题需要具体问题具体分析，读者可以在学习中慢慢积累经验。

最后值得一提的是，如果是分析视频流，可能会在视频分析结果中发现“Incorrect Timestamp”之类的提示，有时候这个提示不一定准确，因为该功能目前只适合分析音频流。

3. 其他技巧

前面我们提到过 sip 和 rtp 两个 Filter，它们可以在 Wireshark 中过滤可见包的内容，其他常用的还有根据来源 IP 地址过滤，如：

```
ip.src == 192.168.1.143
```

根据目的地址过滤，如：

```
ip.dst == 192.168.1.143
```

还可以是多种条件的组合等，如：

```
ip.src == 192.168.1.127 && ip.dst == 192.168.1.143 && udp.port == 5060
```

上面我们抓到的是一个完整的呼叫。有时候，我们可以为了跟踪一路通话的语音问题而从一个通话的中间开始抓包。这时候，由于缺少 SIP 的上下文信息，所有抓包的内容显示为 UDP（即如果没有 SIP 包的话，Wireshark 就认不出这些 UDP 包是 RTP 包了）。如果出现这样的情况，我们需要找到类似 RTP 的包。比方说，如果我们知道呼叫是 PCMU 编码的，那么它的包长可能是 214 字节（IP+UDP 包头 42 字节加上 RTP 包的 172 字节）。找到这类包以后，可以点击某一个包，然后从右键菜单中选择 “ Decode As ... ”，进而选择 RTP，就可以将同类的包用 RTP 包的格式进行解码了。

更多的关于 Wireshark 的使用方法及技巧请参阅相关资料。

10.4 originate 命令实例解析

originate 命令是我们在调试过程中经常使用的一个命令，通过它几乎可以了解到系统中所有的知识，因此我们也在不厌其烦地对其进行讲解。虽然部分内容与前面的章节有所重复，但为了本节的完整性，笔者还是决定多啰嗦几句。另外，这里我们也将进行很多扩展和延伸。

在 FreeSWITCH 控制台上输入 originate 命令后就可以得到一个帮助（后面的小节均根据如下代码讲解）：

```
freeswitch> originate

-USAGE: <call url><exten>|&<application_name>(<app_args>)
        [<dialplan>] [<context>] [<cid_name>] [<cid_num>] [<timeout_sec>]
```

跟其他约定一样，尖括号一般是必选参数，而方括号里面是可选参数。

10.4.1 使用格式和参数

上述代码中，第 1 个参数 call url 就是我们通常所说的呼叫字符串（Dial String）。originate 命令用于从 FreeSWITCH 中向外发起一个呼叫，这个 “外” 就是用这里的呼叫字符串指定的。

关于呼叫字符串的概念我们已经在 4.5.2 节讲过了，这里我们再复习一下。看以下命令：

```
freeswitch> originate user/1000 &echo
```

上述命令中的 user/1000 就是呼叫字符串。它表示从本地的注册用户中查找该用户的联系地址。呼叫字符串的格式是 “类型 / 参数 / 参数”，其中第一部分是字符串的类型。下面我们先来做个实验，看输入一个错误的字符串会出现什么：

```
freeswitch> originate test/1000 &echo
```

```
-ERR CHAN_NOT_IMPLEMENTED

[ERR] switch_core_session.c:496 Could not locate channel type test
[NOTICE] switch_ivr_originate.c:2661 Cannot create outgoing channel of type
[test] cause: [CHAN_NOT_IMPLEMENTED]
```

其中，CHAN_NOT_IMPLEMENTED 表示这种 Channel 的类型没有实现，因而这不是一个合法的字符串，无法创建这种类型的 Channel 并进行呼叫。

那么，系统到底提供了多少 Channel 类型呢？一般来说，每种 Endpoint 都会提供相应的呼叫字符串，每一种呼叫字符串的类型都属于一个 Endpoint Interface，其中一些类型又类似于一种“高级”的呼叫字符串，如 user 和 group。这些呼叫字符串看起来像一个虚拟的 Endpoint Interface，它们最终会解析到底层的 sofia Endpoint Interface。

另外，我们在前面的章节中也讲过，可以通过逗号（,）或竖线符号（|）将多个呼叫字符串隔开，以达到同振或顺振的目的。如下面命令可同时呼叫 1000 和 1001，两个话机都会振铃，哪个先接听则接通哪个，另一路会自动挂断，这种呼叫方式称为“同振”：

```
freeswitch> originate user/1000,user/1001 &echo
```

下列命令就是“顺振”，即第一个号码呼叫失败则呼叫第二个：

```
freeswitch> originate user/1000|user/1001 &echo
```

10.4.2 转入 Dialplan

originate 的第 2 个参数是一个 exten（可以认为是一个分机号），或者是一个“&”符号加上 App。对于后者，我们已经很熟悉了，整个命令的作用是向外发起一个呼叫，建立一个 Channel，对方接听后在本端执行一个 App，如上面的 echo。

对于前者，与上述的情况类似，在向外发起一个呼叫并等待对方接听后，FreeSWITCH 就与对方建立了一个 Channel，Channel 的远端当然是接听电话的用户。在本端与上述 App 的情形不同的是，它会将 Channel 转入 Dialplan 去路由，路由要查找的目的地就是 exten。也就是说，在用户接听后，这种情况跟用户呼入的处理是一样的，都是进入 Dialplan 进行路由，如在默认的配置中，下面的两条命令基本上是等价的：

```
freeswitch> originate user/1000 9196
freeswitch> originate user/1000 &echo
```

前者在 1000 接听后进入 Dialplan，找到 9196 这个 exten，然后再执行 echo。

接下来再看第 3 个参数，它是 Dialplan 的类型，如果不设置，默认就是 XML。我们在第 8 章曾讲到过 inline Dialplan，因此也可以在这里试一下，它与上面的例子也是等价的：

```
freeswitch> originate user/1000 echo inline
```

第 4 个参数是 Dialplan 的 Context，对于 inline Dialplan，它会忽略 Context，而对于 XML 则是有效的，如：

```
freeswitch> originate user/1000 1001 XML public
```

上述命令在 1000 接听后会进入 public Dialplan 查找路由。

10.4.3 更改主叫号码

接下来的两个参数分别是主叫名称（cid_name，Caller ID Name）和主叫号码（cid_number，Caller ID Number）。比如初学者经常看到 0000000000 这样的主叫号码，那是 FreeSWITCH 发起呼叫时默认使用的主叫号码。我们也可以在发起呼叫时自行指定主叫号码，而不使用默认值，如：

```
freeswitch> originate user/1000 &echo XML default 'Seven Du' 7777
```

注意，如果参数中有空格，需要用单引号引起来。上述命令产生的 INVITE 消息如下，其中主叫号码信息在 From 和 Remote-Party-ID 字段中（这里省略了其他字段）。

```
INVITE sip:1000@192.168.1.127:47294;rinstance=a39d7b2af86ea27d SIP/2.0
Via: SIP/2.0/UDP 192.168.1.127;rport;branch=z9hG4bKc0yQ4NU04paXr
Max-Forwards: 70
From: "Seven Du" <sip:7777@192.168.1.127>;tag=y5j452DSQNUBS
To: <sip:1000@192.168.1.127:47294;rinstance=a39d7b2af86ea27d>
Remote-Party-ID: "Seven Du" <sip:7777@192.168.1.127>;party=calling;screen=yes;
privacy=off
```

图 10-8 所示是软电话客户端显示出来的来电信息。

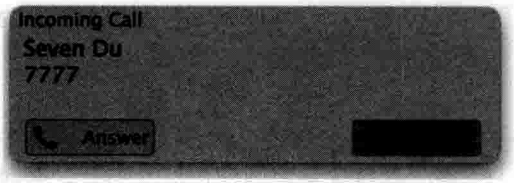


图 10-8 来电时软电话显示的主叫号码

10.4.4 处理呼叫超时

最后一个参数是超时的秒数。值得注意的是，它并不是对方不接电话超时的秒数，而是对方在收到我们 INVITE 消息后，不回复 100 Trying 消息的时间。一般来说这种情况是 IP 地址不可达。如下面的例子中，我们使用了一个不存在的 IP 地址 192.168.100.100：

```
freeswitch> sofia profile internal siptrace on
freeswitch> originate sofia/internal/1000@192.168.100.100 &echo XML default
'Seven Du' 7777 10

[NOTICE] switch_channel.c:1030 New Channel ......

send 1141 bytes to udp/[192.168.100.100]:5060 at 10:21:08.564598:
------------------------------------------------------------------------
INVITE sip:1000@192.168.100.100 SIP/2.0
send 1141 bytes to udp/[192.168.100.100]:5060 at 10:21:09.566271:
send 1141 bytes to udp/[192.168.100.100]:5060 at 10:21:11.567140:
send 1141 bytes to udp/[192.168.100.100]:5060 at 10:21:15.568274:
[NOTICE] switch_ivr_originate.c:3410 Hangup
sofia/internal/1000@192.168.100.100 [CS_CONSUME_MEDIA] [NO_ANSWER]
-ERR NO_ANSWER
```

上面我们先打开了 siptrace，从输出的信息中我们可以看出，在 FreeSWITCH 发出 INVITE 消息后，由于没有收到 100 Trying 回复，于是在 1 秒后重发 INVITE 消息，如果还收不到则于 2 秒、4 秒后重发⊖，由于我们指定了 10 秒超时，因此该呼叫于 10 秒后失败，

⊖ 关于重发隔可以参考 RFC3261，也可以把这里的超时时长改大一点，如 100，自己跟踪消息看一下。

返回 NO_ANSWER。

10.4.5 防止命令阻塞

originate 命令是阻塞的，因此如果执行上述命令，则无法输入其他命令或取消该呼叫。解决这一问题有两个办法：

- 使用 bgapi，如 bgapi originate sofia/......
- 开启另外一个 fs_cli 客户端，使用 show channels 找到该呼叫的 UUID，然后执行 uuid_kill <uuid>；或者，直接执行 hupall 挂断所有通话。

以上是在命令行上的解决办法，尤其要注意的是，如果我们在 Event Socket 方式下使用 originate 发起呼叫，一般要使用 bgapi 来避免阻塞，如：

```
freeswitch> bgapi originate user/1000 &echo
```

10.4.6 使用通道变量

大家已经知道，通道变量可以影响呼叫的行为。我们在 orignate 时也可以使用通道变量。到这里，我们又回到呼叫字符串，因为通道变量是加在呼叫字符串上的。

通过使用通道变量，下列命令也能改变主号名称和号码：

```
freeswitch> originate {origination_caller_id_name='Seven Du',
originatioin_caller_id_number=7777}user/1000 &echo
```

多个通道变量之间使用逗号隔开，有时候有的通道变量里会有逗号，可能造成冲突，因而下面的命令都达不到你想要的效果：

```
freeswitch> originate {absolute_codec_string=G729,PCMU}user/1000 &echo
freeswitch> originate {absolute_codec_string='G729,PCMU'}user/1000 &echo
```

上述命令的本意是在呼叫时，在 SDP 里向对方提供 G729 和 PCMU 编码，但我们执行上述命令，却在日志中看到 PCMU 被丢掉了：

```
freeswitch> originate {absolute_codec_string='G729,PCMU'}user/1000 &echo
[DEBUG] switch_ivr_originate.c:2060 Parsing global variables
[DEBUG] switch_event.c:1615 Parsing variable [absolute_codec_string]=[G729]
```

所以，我们需要对 Codec 字符串里的逗号进行转义，可以使用一个反斜杠来进行转义，如：

```
freeswitch> originate {absolute_codec_string=G729\,PCMU}user/1000 &echo
```

或者，使用“^^”进行转义，用别的符号代替逗号分隔符，如：

```
freeswitch> originate {absolute_codec_string=^^:G729:PCMU'}user/1000 &echo
```

其中，“^^”后面跟冒号表示以后要用冒号代替逗号，FreeSWITCH 在遇到冒号时就会把它用逗号替换回来，这样就可以避免因为逗号冲突而导致错误[⊖]。

⊖ 在使用 sip_h_、sip_rh_ 及 sip_ph 时，这种转义操作不适用，所以还得用反斜杠来解决，如 {sip_h_X-My-Header=one\,two\,three}。

有时候，这些通道变量参数可能是程序里自动生成的，因而计算何时该用逗号可能要多费一些代码，所以FreeSWITCH也支持将不同的变量写到不同的大括号中，如：

```
freeswitch> originate {var1=1}{var=2}{var3=3}user/1000 &echo
```

还有一类通道变量是在方括号中定义的，称为局部通道变量。**它唯一作用于某一条腿**。在先呼叫某一用户的办公分机时，如果超时，则通过网关gw1呼叫其手机号，如果再超时，则通过网关gw2呼叫：

```
freeswitch> originate {var1=1}[leg_timeout=10]user/1000|
    [leg_timeout=20]sofia/gateway/gw1/1380000000|
    [leg_timeout=20]sofia/gateway/gw2/1380000000
```

其中，leg_timeout只应用于靠近它的那一条腿上，它用于定义呼叫超时，即等待对方返回媒体（如183或200）的超时时间。而var1=1是全局的，每条腿都会有这个变量。

通过使用局部的通道变量，就可以为不同的腿（不同的网关或不同的呼叫路径）定义不同的参数，如这里使用leg_timeout为不同的腿定义不同的超时时间。

10.4.7 Early Media对呼叫的影响

一般而言，如果我们呼叫本地的IP话机或软电话，对方回的是180 Ring，同样情况下如果通过网关呼叫PSTN用户的话，则PSTN网络往往会返回Early Media，也就是前期的振铃音或彩铃。PSTN在呼叫失败时也会返回Early Media，如“您拨的电话忙，请稍后再拨……”等。Early Media一般是用183消息指示的。因而，一旦对方返回了183，FreeSWITCH就认为originate成功了。注意，从这里可以看出，前面我们讲到：“在执行originate时，originate命令会一直等待对方接听才返回”，这种说法不是很准确，当时我们只是为了讲起来简单。严格来说，originate命令是在收到媒体指示就返回的，如收到SIP中的183或200消息。一般来说，软电话不会返回183，因而该命令直到软电话接听（即收到200时）才返回，但在通过外部网关拨打PSTN电话的时候往往不是如此，PSTN网络中大多数情况下会在电话应答前返回回铃音（即Eearly Media），然后相连的网关设备就会向FreeSWITCH回复带SDP的183消息，以建立媒体通路。

有时候，有些Early Media对我们没有意义，如我们在主动外呼的应用中（如电话自动催费）希望用户真正接听以后才对他放音，或进入IVR，这时怎么办？有一个参数可以帮我们做到，如下：

```
freeswitch> originate {ignore_early_media=true}sofia/gateway/gw/1380000000
    &playback(/tmp/test.wav)
```

通过指定ignore_early_media变量，FreeSWITCH在发起呼叫时会忽略对方返回的Early Media，直到等到收到正常的应答信号（如200 OK）才返回。

10.4.8 bridge也使用originate

读到这里读者可能困惑了，为什么bridge也使用originate？何况我们讲过，bridge是一

个 App 而 originate 是一个 API？

带着这个问题我们来看下面这个例子。我们使用 originate 命令发起一个呼叫，接通 1000 和 1001 两个号码：

```
freeswitch> originate user/1000 &bridge(user/1001)
```

当上述 originate 命令发起呼叫时，建立一个 Channel，然后呼叫 1000，1000 接听后，在该 Channel 上（它相当于 a-leg）执行 bridge App，bridge 会再建立另一个 Channel（即 b-leg）来呼叫 1001。此时，1000 和 1001 这两个 Channel 在信令上建立了桥接关系。1001 接听后，bridge 会把它们的媒体也桥接起来，此时进入正常通话。

实际上，在 FreeSWITCH 内部，originate 和 bridge 都是调用同一个函数来实现的，为了简单起见，我们使用如下伪代码：

```
originate(session, new_session, dial_string);
```

当 originate 调用该函数时，session 为空，因此相当于调用：

```
originate(NULL, new_session, "user/1000");
```

这时它就建立了一个新的 session，即 new_session，我们称之为 session_a。这里 user/1000 为呼叫字符串，如果该呼叫字符串包含有 {originattion_caller_id _number=1000}，则会以 1000 作为主叫号码（由于主、被叫号码相同，看起来相当于 1000 呼 1000），否则使用默认的 00000000 作为主叫号码。

如果 a 接听了，在调用 bridge 时，bridge 会进行如下函数调用：

```
originate(session_a, new_session, "user/1001");
```

上述函数调用也会建立一个新的 new_session（b-leg）来呼叫 1001，但与第一次调用不同的是，这次它有参照物了。这个参照物就是我们刚才建立的 session_a。比如，它会使用从 session_a（a-leg）中获得的主叫号码作为 b-leg 的主叫号码，它会根据 a-leg 上的 call_timeout 参数来决定 b-leg 的呼叫超时时长等。

总之，a-leg 和 b-leg 现在已经是建立关系了，关于呼叫接下来的进展我们在下一节讨论。

10.4.9 bridge 中的 Early Media

在本节我们通过 Early Media 的例子来进一步讲解 bridge。

在上节所述的场景下，如果 b-leg 返回 SIP 的 183 Session Progress，则 FreeSWITCH 也会向 a-leg 发 183 Session Progress，因此可以将 b-leg 上收到的媒体发到 a-leg 上，a-leg 就听到了回铃音（或彩铃、呼叫失败提示音等）。

如果 b-leg 返回 180，则由于 a-leg 已经接听（a-leg 已经处于 answered 状态，即处于正常通话状态），因此 FreeSWITCH 无法为 a-leg 发送 180 Ringing 消息，因而 a-leg 在 b-leg 振铃期间将听不到任何声音。

为了解决这个问题，可以让 FreeSWITCH 作为一个中间人，为 a-leg 在这种情况下产生一个假的回铃音，实现方法是在 bridge 之前设置一个 transfer_ringback 变量，如我们需要使

用下面的方法重新发起呼叫：

```
freeswitch> originate {transfer_ringback=local_stream://moh}user/1000
&bridge(user/1001)
```

这时 a-leg 接听后，如果 b-leg 振铃，a-leg 将能听到假的回铃音。这里假的回铃音是在收到 b-leg 的 180 时开始播放的。有时候，等待收到 b-leg 的 180 Ringing 也需要很长时间，因而另外一个参数 instant_ringback 可以让 bridge 立即播放回铃音：

```
freeswitch> originate {instant_ringback=true}{transfer_ringback=local_
stream://moh}user/1000 &bridge(user/1001)
```

以上的参数都是用在 a-leg 已经接听的情况，即回呼的情况。关于 a-leg 正常呼入的情况我们将在下一节介绍。

10.4.10　bridge 中的主叫号码

关于主叫号码的显示（俗称来电显示）是大家普遍关注的一个问题。本节就带领大家了解一下和主叫号码显示的知识。在 10.4.4 节我们讲过，使用 FreeSWITCH 发起呼叫时可以使用下面的命令设置主叫号码（下面是将主叫号码设为 7777）：

```
freeswitch> originate {origination_caller_id_number=7777}user/1000 &echo
```

设置好以后，假设我们使用 bridge 呼叫 b-leg，具体实现如下：

```
freeswitch> originate {originattion_caller_id_number=7777}user/1000 &bridge(user/1001)
```

由 FreeSWITCH 发现 a-leg 是一个回呼，当 a-leg 接听后，再作为主叫去呼叫 b-leg 时，会进行主叫号码翻转。在日志中我们就可以看到如下的行：

```
[INFO] switch_channel.c:2978 sofia/internal/sip:1000@192.168.1.127:47294
Flipping CID from "" <7777> to "Outbound Call" <1000>
```

由以上信息可以知道，翻转后 a-leg 的主叫号码就变成了 1000，它去桥接 1001 时，显示的主叫号码 1000 是正确的。

但是有的时候，我们想改变 1000 这个主叫号码（当然这里我们不管是出于什么目的，这里仅进行技术上的讨论），比方说我们想把它变成 8888，那么可以使用如下的方法（注意，因为我们这里不关心 a-leg 在接到电话时显示什么主叫号码，所以省略一些参数，命令行短了一些）：

```
freeswitch> originate user/1000 &bridge({origination_caller_id_number=8888}
user/1001)
```

与 originate 类似，bridge 的参数也是一个呼叫字符串（从 10.4.8 节我们知道，bridge 本质上就是 originate），因而它也可以使用一样的通道普量以控制主叫号码的显示。

如果直接在 b-leg 上设置变量不方便，那么改变 b-leg 上显示的主叫号码还可以在 a-leg 上做文章。回想一下，bridge 在调用 originate 进行呼叫的时候是知道 a-leg 的信息的，这个信息的获取就来自于 a-leg 的 effective_caller_id_number 这个变量，所以我们可以通过以下命令达到这个目的：

```
freeswitch> originate {effective_caller_id_number=8888}user/1000 &bridge(user/1001)
```

总之 effective_caller_id_number 变量设置在 a-leg 上，但影响 b-leg 的主叫号码显示（俗称来电显示）；origination_caller_id_number 可以设置到 a-leg，也可以设置到 b-leg 上，它将影响本 leg 的来电显示。

最后我们把这个例子做完整。我们使用命令呼叫 1000 和 1001 两个号码，在 1000 上显示 7777，在 1001 上显示 8888。下面两种方法是等价的：

```
freeswitch> originate {originatioin_caller_id_number=7777}user/1000
     &bridge({origination_caller_id_number=8888}user/1001)

freeswitch> originate {originatioin_caller_id_number=7777}{effective_caller_
id_number=8888}user/1000
     &bridge(user/1001)
```

至此，我们对 originate 和 bridge 的讨论就告一个段落了。真正理解和掌握这些内容还需要读者多做练习。

10.5 呼叫是怎样工作的？

本节我们分析一个呼叫在 FreeSWITCH 里面到底是怎么工作的，若涉及前面的一些基础概念，我们也会简单复习一下，复习不到的请自行翻阅前面的章节。

首先，我们假设你用的是默认的配置，并从 1000 呼叫 1001。我们把 1000 称为主叫，把 1001 称为被叫，如图 10-9 所示。

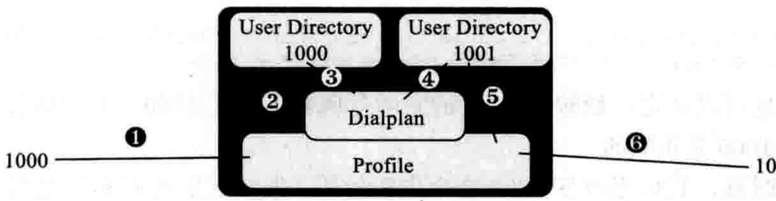


图 10-9 FreeSWITCH 中的呼叫流程

1000 的 SIP 话机作为 UAC 会发送 INVITE 请求到 FreeSWITCH 的 5060 端口（见图 10-9 中的①），也就是到达 mod_sofia 的 internal 这个 Profile 所配置的 UAS（见 conf/sip_profiles/internal.xml）。该 UAS 收到正确的 INVITE 后会返回 100 响应码，表示我收到你的请求了，稍等片刻。该 UAS 中对所有收到的 INVITE 都要进行鉴权（因为 auth-calls=true）。它会先检查 ACL（访问控制列表。一般用于 IP 鉴权，我们暂时还没学到 ACL，留到后面去讲）。默认 ACL 检查是不通过的，因此就会走到密码鉴权阶段，SIP 没有发明新的鉴权方式，而是使用与 HTTP 协议中一样的 Digest Auth [⊖]进行鉴权。其特点就是密码并不在网络上传输，因而比较安全。读者可以翻到第 7 章看一下具体的认证流程，一般是 UAS 回复

⊖ http://en.wikipedia.org/wiki/Digest_access_authentication。

401，然后 UAC 重新发送带鉴权信息的 INVITE。UAS 收到后，便将鉴权信息提交到上层的 FreeSWITCH 代码，FreeSWITCH 就会到 Directory（用户目录）中查找相应的用户（见图 10-9 中的②），在这里它会找到 conf/directory/default/1000.xml 文件中配置的用户信息，并根据其中配置的密码信息进行鉴权。如果鉴权不通过，则回送 403 Forbidden 等错误信息，会话结束。

如果鉴权通过，FreeSWITCH 就取到了用户的信息，比较重要的是 user_context，在我们的例子中它的值为 default。接下来电话进入路由（Routing）阶段，开始查找 Dialplan（见图 10-9 中的③）。由于该用户的 Context 是 default，因此路由就从 default 这个 Dialplan 查起（由 conf/dialplan/default.xml 定义）。Context 的对应关系如图 10-10 所示。

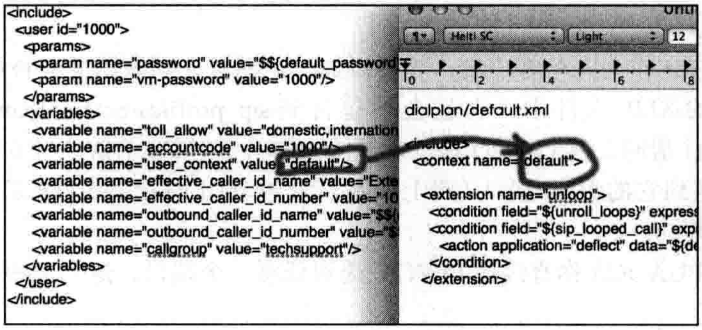


图 10-10　Context 的对应关系

查找 Dialplan 的流程已经在第 6 章讲过了，总之它会帮你找到 1001 这个用户，并执行 bridge user/1001。这里，user/1001 称为呼叫字符串，它会再次查找 Directory，找到 conf/directory/default/1001.xml 里配置的参数（见图 10-9 中的④）。由于这里 1001 是被叫，因此它会进一步查找直到找到 1001 实际注册的位置。实际上，它实际的注册位置是用 dial-string 参数来表示的，由于所有用户的规则都一样，因此该参数被放到 conf/directory/default.xml 中，在该文件中可以看到如下设置：

```
    <param name="dial-string" value="{sip_invite_domain=${dialed_domain},presence_
id=${dialed_user}@${dialed_domain}}$
        {sofia_contact
        (${dialed_user}@${dialed_domain})}"/>
    </params>
```

其中，最关键的是 sofia_contact 这个 API 调用，它会查找数据库，找到 1001 实际注册的 Contact 地址，并返回真正的呼叫字符串。如在笔者的机器上执行如下命令可以返回 1001 注册的 Contact 地址：

```
freeswitch> sofia_contact 1001@192.168.1.109

sofia/internal/sip:1001@192.168.1.109:51641
```

找到呼叫字符串后，FreeSWITCH 又启动另外一个会话（见图 10-9 中的⑤）作为

一个 UAC 给 1001 发送 INVITE 请求（见图 10-9 中的⑥），如果 1001 摘机，则 1001 向 FreeSWITCH 回送 SIP 200 OK 消息，FreeSWITCH 再向 1000 回 SIP 200 OK，通话开始。

FreeSWITCH 是一个 B2BUA，上面的过程建立了一对通话，其中有两个 Channel。

参照第 7 章，读者不妨跟踪 SIP 消息试一下，在控制台上打开跟踪的命令是：

```
sofia profile internal siptrace on
```

关掉调试的命令是：

```
sofia profile internal siptrace off
```

在 FreeSWITCH 的默认配置中，external.xml 对应的 Profile 是不鉴权的，因此凡是送到 5080 端口的 INVITE 都不需要鉴权。那么什么情况下或者怎样将 INVITE 会送到 5080 呢？有以下两种情况：

- ❑ FreeSWITCH 作为一个客户端，若要添加一个网关，则该网关会被放到 sip_profiles/external/ 的 XML 文件中，它就会被包含到 sip_profiles/external.xml 中。它向其他的服务器注册时，其 Contact 地址就是 IP:5080，如果有来话，对方的服务器就会把 INVITE 送到它的 5080 端口（跟上面我们呼叫 1001 时把 INVITE 送到 51641 端口是同样的道理）。
- ❑ 大部分 SIPUA 允许你直接把 INVITE 送到任意一个端口，这一般用在中继方式对接的情况。

读者也可以看到，在 external.xml 中 auth-calls=false。所以，当它收到 INVITE 时就不会进行鉴权了，而是直接对来话进行路由。

而路由需要一个 Context，这个 Context 从哪里来呢？仔细看一下 external.xml，里面有这样一条：

```
<param name="context" value="public"/>
```

所以路由就会查找 public 这个 Dialplan，而其中只定义了很少的 Extension，也就是说不经过认证的 INVITE 请求一般只能做比较少的事情，如只处理来话。

读到这里，读者也许会问：我怎么看到 internal.xml 里的 context 值也是 public 呢？

不错，但是 internal 这个 Profile 中还有 auth-calls=true 这一参数。对于一路来话而言，如果经过了认证，就会找到一个用户，然后就会使用我们最前面所说的用户目录中的 user_context，而不是使用这个 Profile 中的 context。即用户目录中的 user_context 比 Profile 中的 context 参数优先级要高。

那么，这个 Profile 中的 context 什么时候用呢？

前面已经说过我们还没讲到 ACL，不过，如果你正确配置了 ACL 的话，对有些 IP 地址发来的 INVITE 请求可以不鉴权，那么自然就不知道是哪个用户打来的电话，因此就无法找到对应的 user_context，那就只能使用 Profile 提供的 context 了。当然，你可以根据需要把这个 context 改成任意值，只要你配置了相应的 Dialplan 进行路由就行了。

值得一提的是，对于没有经过鉴权的呼叫，同样可以有主、被叫号码，只不过主叫号码

是不可信的，对方可以指定显示任意号码。如果你讲究严谨的话，就需要多做些工作了，如使用我们 10.4.13 节说的 effective_caller_id_number 强制设置显示某些主叫号码等。

10.6 FreeSWITCH 图形用户界面简介

图形用户界面即我们通常所说的 GUI（Graphic User's Interface）。对于广大开发者和极客们来讲，使用命令行配置和手工修改配置文件是最理想的选择。但对于 FreeSWITCH 新手或维护人员来讲，可能更喜欢一个图形界面来帮助理解和使用 FreeSWITCH。

对于 GUI 的开发来讲，FreeSWITCH 提供了丰富的底层接口支持，因而很容易在上层进行开发。但是，开发一个功能全又适合所有人胃口的 GUI 也绝非易事。FreeSWITCH 官方的 wiki ⊖上列举了一些现有的 GUI 产品，有开源的也有商业的，它们各有特点，但到目前为止还没有一个占统治地位的 GUI 产品。当然，出现这一问题的原因也可能是 FreeSWITCH 提供的接口和 API 比较丰富，因而很容易集成到现成的业务系统中去，因此开发通用 GUI 的需求不是很强烈。

一般来说，开发 FreeSWITCH 的 GUI 有两种方法：

- 通过提供图形界面，后台修改 FreeSWITCH 的 XML 配置文件，并调用 reloadxml 或其他重启指令使之生效。这种方法只适合管理一台 FreeSWITCH，即适合少量用户的管理。
- 通过使用 FreeSWITCH 提供的 xml_curl 接口，提供一个 HTTP 服务器动态向 FreeSWITCH 提供 XML。这种方法可以管理多台 FreeSWITCH，并适合大规模部署。

下面我们来介绍几款典型的开源 GUI 产品。

10.6.1 FusionPBX

FusionPBX 是使用 PHP 开发的 FreeSWITCH GUI。它支持 Linux、Windows、BSD、Mac OS X 等多种操作系统，并支持 PostgreSQL、MySQL、SQLite 等数据库。

它是通过将数据存储到数据库中，并修改本地的 XML 配置文件实现的。

它支持无限的分机、IVR、呼叫组、传真、会议、队列、呼叫前转、点击拨号，以及实时呼叫状态显示等诸多特性，使用起来比较方便。

FusionPBX 一直处于比较活跃的开发中。更多的介绍请参考 http://www.fusionpbx.com。

10.6.2 blue.box

blue.box 是 2600Hz 团队开发的一款开源的 GUI 产品，使用 PHP/MySQL 技术开发。它最初的开发是基于著名的 Asterisk 的 GUI FreePBX，号称是 FreePBX V3。它也是使用修改本地 XML 配置文件并重新加载的方式工作。除支持普通的用户、分机、路由管理外，它还支持多语言和多租户。

⊖ http://wiki.freeswitch.org/wiki/Freeswitch_Gui。

由于 2600Hz 团队近几年一直把精力放在基于 Erlang 的云平台上，所以 blue.box 的开发一度中断。不过，在 ClueCon 2013 上，2600Hz 团队又宣布了该项目的重生，新的开发计划将基于 Python。

更多的介绍请参考：http://www.2600hz.org。

10.6.3 FreeSWITCH Portal

FreeSWITCH-Portal 是笔者发起的一个项目。与其他 GUI 系统不同的是，该项目并不致力于替代上述的 GUI 系统，而是作为它们的一个补充，提供一些开发者常用的功能，便于新手进行学习。

首先，FreeSWITCH Portal 提供了开箱即用（Out of the Box）功能。上述的各种 GUI 产品都或多或少依赖于 Apache/Nginx 等 Webserver、PHP 等开发语言及 PostgreSQL/MySQL 等数据库，因此，配置 GUI 本身就需要一定的知识和基础。而 FreeSWITCH Portal 内置于 FreeSWITCH，它仅依赖于 mod_xml_rpc 模块提供的内置于 FreeSWITCH 的 Webserver。该模块是默认编译的，但不加载，因此用户只需要加载该模块即可使用它。

在 FreeSWITCH 控制台上执行下列命令就可以加载该模块：

```
freeswitch> load mod_xml_rpc
```

如果想让该模块随着 FreeSWITCH 启动而自动加载，可以在 conf/autoload_configs/modules.xml 中将模块的注释去掉，去掉注释后的配置如下：

```
<load module="mod_xml_rpc"/>
```

然后使用浏览器打开 http://localhost:8080/portal（如果 FreeSWITCH 所在主机与要访问的电脑不同，则将 localhost 换成 FreeSWITCH 所在主机的 IP 地址），如果提示分别输入用户名和密码则分别输入“freeswitch”和“works”，该密码可以在 conf/autoload_configs/xml_rpc.conf.xml 中配置。系统界面如图 10-11 所示。

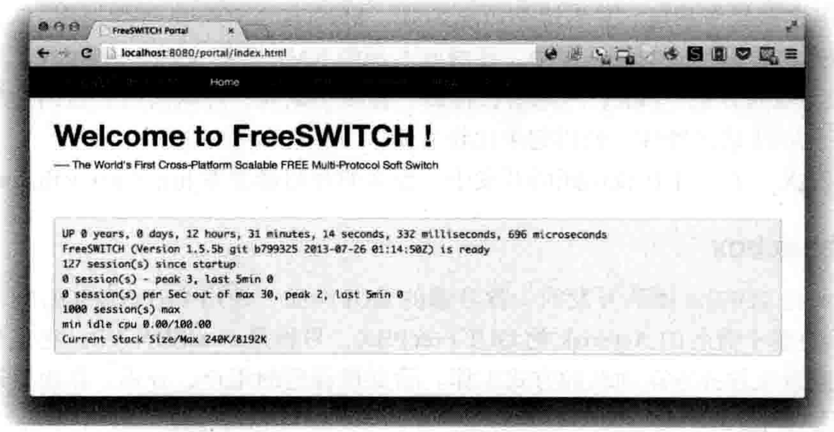


图 10-11 FreeSWITCH Portal

系统主菜单一共可以分为以下几个部分：

- Users：显示所有用户。
- Calls：显示当前的呼叫状态，如果选中“Auto Update”复选框，则表示它是实时更新的。
- Channels：显示当前所有 Channel 的状态，它是实时更新的。
- Show：调用系统的 show 命令显示很多 FreeSWITCH 内部的东西，如注册信息、已加载的模块、App、API 等。

如果看到绿色的“Socket Connected”，则表示 Websocket 连接正常。注意，目前 Websocket 是不认证的，因此可能有安全性问题，如果你在生产系统中使用，请注意安全。

如果你看到右上角红色的“Socket Disconnected”也不用担心，那是因为你的系统不支持 Websocket，修改 xml_rpc.conf.xml，将“enable-websocket”参数设为 true 就可以了。

```
<param name="enable-websocket" value="true"/>
```

另外，你也需要使用能支持 Websocket 的浏览器，现代的浏览器如 Chrome、FireFox、Opera、IE9 及以上版本等都支持 Websocket。

如果你的系统不支持 Websocket，你也不会损失太多功能。数据的更新会自动变成通过 Ajax 轮循方式获取。

另外，如果你是一个开发者的话，相信你比较关心系统的事件。我们在 5.1.1 节的最后讲过在 fs_cli 中订阅和查看事件的方法，但是好多人会让 FreeSWITCH 动辄数百行的“大”事件吓倒。在 FreeSWITCH Portal 里，笔者还留了一个“后门”，可以允许用户用更好的方法查看系统事件。

很多浏览器（如 Chrome 或 FireFox）都具有网页调试功能（FireFox 上有著名的 FireBug 插件）。我们这里以 Chome 为例来说明。如图 10-12 所示，在 Chrome 浏览器页面上右击，单击最下方的“审查元素”(Inspect Elements)，即可在页面下方打开调试窗口。

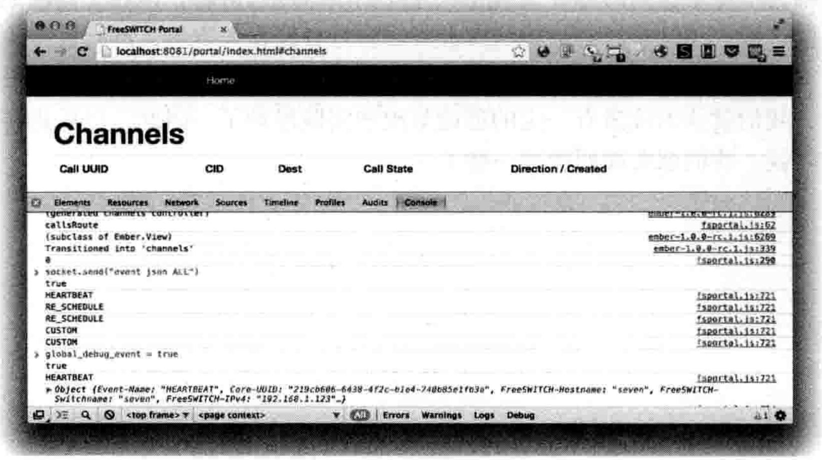


图 10-12　通过浏览器的调试功能查看系统事件

在调试窗口中切换到 Console 页，输入：

```
event("ALL");
```

即可订阅全部事件。如果这时候打个电话，就可以看到调试窗口中输出各种事件的名字，读者就知道打一个电话大体都会产生哪些事件了。细心的读者也可以做一下“慢动作”，顺便观察不同时间产生的不同事件，如振铃时产生什么事件、接听时产生什么事件、挂机后又产生什么事件等。

另外，通过打开全局的调试事件的选项可以看到更详细的事件信息，在调试窗口中输入：

```
global_debug_evnet = true;
```

如果不想看详细事件了，则把上述变量改成 false 即可，或直接刷新页面。

当然，任何时候都可以通过 api 或 bgapi 函数来直接执行 FreeSWITCH 的 API 命令，如：

```
api("status");
api("sofia status");
bgapi("originate user/1000 &echo");
```

总之，FreeSWITCH Portal 致力于简单、实用、易用。如果读者发现上面提到的某些功能不好用，可以更新到最新的版本（2013 年 8 月 14 日以后的版本）。在该系统的 About 页面上，也列了一些将要完成（Todo）的功能，如增加国际化（如中文）语言的支持等，如果读者有兴趣的话也欢迎贡献自己的力量。

10.7 小结

本章列举了一些在实战过程中常用的基本技能、调试技巧和实用工具等。这些工具，讲起来都很简单，但在实际使用过程中，需要多多练习才能掌握其要领。熟练使用这些命令和工具，在以后的学习和工作中将能事半功倍。

限于篇幅，本章中提到的部分工具和软件我们没有深入讲解，但在关键的地方，笔者都给出了相关的链接或关键字，有兴趣的读者可以自行查阅相关资料，进行进一步的学习。

另外，本书前面的章节对于各种命令和使用场景都讲得比较详细（因而显得有点啰嗦）。从本章开始，我们就认为读者有一定的理论基础和实践经验了。因此，以后再讲解配置、命令或代码的时候，我们就要简明扼要一些了。

第 11 章 基本功能与实现

FreeSWITCH 的功能非常丰富。本章我们带领大家通过一些实际的例子来学习 FreeSWITCH 的基本功能，并在实践中深入领会各种配置文件的意义。

11.1 批量创建用户

如何在 FreeSWITCH 中快速创建大量用户是在社区中经常遇到的一个问题。其实该问题解决起来并不难。我们已经了解现有的用户配置文件是存放在 FreeSWITCH 安装目录的 conf/directory/default 目录下，每个用户对应一个 XML 配置文件，如 1000.xml 包含了 1000 这个用户的配置文件。FreeSWITCH 默认提供了 1000 ~ 1019 这 20 个用户，如果要手工创建另外一个用户，如 1020，我们只需以 1000.xml 为模板，将该文件中的内容复制到 1020.xml，然后把 1020.xml 文件中的所有出现 1000 的地方全部替换成 1020 即可（这也是我们在 3.3 节使用的方式）。

当然，除了手工的复制和替换外，上述步骤可以在 UNIX 系统上的 Shell ⊖中使用下列命令完成：

```
# sed -e "s/1000/1020/" 1000.xml > 1020.xml
```

其中 sed 是 UNIX 系统上经典的流文件编辑器，上面我们使用 sed 的“s”命令将 1000.xml 文件中所有出现 1000 的地方都替换成 1020，然后将命令的输出重定向（大于号是 Shell 中的重定向操作符）到 1020.xml 文件中。

上述命令用于创建一个用户的情况，如果要创建一批用户怎么办呢？其实也很简单，我们只需要用一下 Shell 中的 for 循环就可以了。比如我们要创建 1020 ~ 1039 这 20 个用户，具体的 Shell 命令如下：

⊖ 这里我们假定使用 Bourne Shell 系列的 Shell。UNIX 历史上有多种 Shell，最典型的是 Bourne Shell 和 C Shell 两大类，它们的语法稍有不同。目前使用比较多的都是 Bourne Shell 的变种，如 Burne-Again Shell（即 bash）、zsh 等。

```
# for i in `seq 1020 1039`; do sed -e "s/1000/$i/" 1000.xml > $i.xml ; done
```

在上述命令中，首先，seq ㊀命令会产生一个序列，两个参数分别是序列的起止点，如：

```
$ seq 1 3
1
2
3
```

有了这个序列以后，for 循环就可以从该序列中依次取得每一个值，并将该值赋值给“i”这个变量。do 和 done 之间是循环体，其中的内容就是我们前面的 sed 命令，只不过我们使用“$i”这个变量引用，它的值都是每次循环时从序列中取得的。注意其中的分号也是有用的，它用于在同一行上隔开多个命令。详细的 Shell 语法我们就不多讲了，总之用上述命令批量生成用户是很方便的。

另外，在 Mac 平台上，笔者发现竟然没有 seq 这个命令，只好自己用 Ruby 写了一个。下面的 Ruby 脚本与 Linux 上的 seq 功能相似，同样，详细原理我们就不多解释了。

```
#!/usr/bin/env ruby
(ARGV[0].to_i .. ARGV[1].to_i).each { |x| puts x}
```

当然，好多读者也在使用 Windows 平台。但很遗憾的是，Windows 上的 Shell 功能比较弱，也没有像 sed 之类的工具，因此做起来不如在 UNIX 上方便。不过，也有人把经典的 UNIX 工具软件都移植到了 Windows 上，称为 UnxUtils ㊁。安装完 UnxUtils 后，就可以使用大部分的 UNIX 命令了。如我们可以使用如下“.bat”脚本完成同样的添加用户的功能（将下列内容存到与 1000.xml 相同的目录下的扩展名为“.bat”的文件中，用鼠标双击即可执行）：

```
for /L %%i in (1020, 1 1039) do sed -e "s/1000/%%i" 1000.xml > %%i.xml
```

除此之外，在 FreeSWITCH 源代码目录中的 scripts/perl 目录下也有一个 add_user 脚本，该脚本是用 Perl 写的，也能完成类似的功能，具体命令如下：

```
# ./add_user 1020
```

上述命令将会把用户配置文件安装在默认的安装路径下面（使用默认的 Domain——default）。当然，也可以在命令行上改变 Domain 和安装路径，如下列命令可以将配置文件安装在 /opt/freeswitch/conf 目录中的 directory/my_domain 目录下：

```
# ./add_user 1020 --domain=my_domain --confpath=/opt/freeswitch/conf
```

也可以使用“--users”参数指定批量创建的用户的范围，如：

```
# ./add_user --users=1020-1039
```

总之，如果不确定上述脚本的使用方法，最好先试验一下。比如，我们先创建一个新的目录用于测试：

㊀ 注意，括在 seq 外面的不是单引号，而是重音符（“`”），即键盘上数字“1”左边那个按键。参见：http://zh.wikipedia.org/wiki/ 重音符。

㊁ 安装方法可以参考 http://www.dujinfang.com/2010/05/27/zai-windows-shang-an-zhuang-unixutils.html。

```
# mkdir -p /tmp/directory/my_domain
```

然后使用下列命令可以看出它确实能创建指定的配置文件：

```
# ./add_user 1111 --users=1020-1039 --domain=my_domain --confpath=/tmp

Added 1020 in file /tmp/directory/my_domain/1020.xml
Added 1021 in file /tmp/directory/my_domain/1021.xml
Added 1022 in file /tmp/directory/my_domain/1022.xml
Added ...
```

需要注意的是，add_user 脚本创建的用户配置文件不是以 1000.xml 为模板的，而是内置在脚本中，因此，如果需要创建的用户与 add_user 脚本中的模板内容不同，需要事先更改脚本中的内容。

总之，不管使用什么方法，创建完用户配置文件后就可以在 FreeSWITCH 中使用 reloadxml 命令使之生效了。在配置生效后，使用这些用户进行注册和呼出操作都没有问题，如果这些用户也需要被叫，那就需要修改 Dialplan，增加到这部分用户的路由。比如，我们增加了 1020 ~ 1039 这 20 个用户后，可以简单地将默认的 Dialplan 中的正则表达式“^(10[01][0-9])$”改为“^(10[0-3][0-9])$”，这样就可以包含我们新创建的用户了。更改后的部分内容如下：

```
<extension name="Local_Extension">
<condition field="destination_number" expression="^(10[0-3][0-9])$">
```

总之，讲起来可能比较啰嗦，但实际使用起来还是挺简单的，读者不妨自己练习一下。

11.2　用 FreeSWITCH 实现 IVR

IVR（Interactive Voice Response，交互式语音响应）实际上就是我们经常说的电话语音菜单。在练习 IVR 的配置之前，我们先来一个感性的认识：FreeSWITCH 默认的配置已包含了一个功能齐全的例子。随便拿起一个分机，拨 5000，就可以听到菜单提示了。当然，默认的提示是英文的，大意是说欢迎来到 FreeSWITCH，按 1 进入 FreeSWITCH 会议；按 2 进入回音（echo）程序（这时候可以听到自己的回音）；按 3 会听到等待音乐（Music on Hold，MOH）；按 4 转到 FreeSWITCH 开发者 Brian West 的 SIP 电话上；按 5 会听到一只猴子的尖叫；按 6 进入下级菜单；拨 9 重听；拨 1000 ~ 1019 之间的号码则会转到对应分机。

有条件的读者可以自行测试一下。有了这些直观的感受之后，我们就可以接着进行下面的配置练习了。

11.2.1　最简单的菜单

我们先来配置一种最简单的情形。一些廉价的小企业交换机通常会提供这样的功能，当有电话呼入时，会播放：“您好，欢迎致电某某公司，请直拨分机号，查号请拨 0”。然后，来话用户就可以输入 1000 ~ 1019 之间的分机号，也可以直接按 0 转到人工台（如分机 1000）进行查号，或要求转接其他分机。

IVR 系统默认的配置文件为 conf/autoload_configs/ivr.conf.xml，它装入 conf/ivr_menus/ 目录下所有的 XML 文件。系统有一个示例的 IVR 配置，叫 demo_ivr，也就是我们刚才拨 5000 听到的那个。

真正的菜单配置信息放到一对“<menus></menus>”标签中，每一对“<menu></menu>”标签就描述一个菜单。每个菜单应该有一个唯一的名字（name），以便在拨号计划（Dialplan）中引用（作为 ivr App 的参数）。在前面的章节中已经讲过，被装入的 XML 文件最外层一般都有一对“<include>”标签，以保证 XML 文档的完整性。为了实现目标菜单，我们创建一个 XML 配置文件 conf/ivr_menus/welcome.xml，内容如下：

```
<include>
    <menus>
        <menu name="welcome"
            greet-long="welcome.wav"
            greet-short="welcom_short.wav"
            invalid-sound="ivr/ivr-that_was_an_invalid_entry.wav"
            exit-sound="voicemail/vm-goodbye.wav"
            timeout="15000"
            max-failures="3"
            max-timeouts="3"
            inter-digit-timeout="2000"
            digit-len="4">

            <entry action="menu-exec-app" digits="0" param="transfer 1000 XML default"/>
            <entry action="menu-exec-app" digits="/^(10[01][0-9])$/"
                   param="transfer $1 XML default"/>
        </menu>
    </menus>
</include>
```

在上述配置中，首先，我们指定菜单的名字（name）是 welcome，其他各项的含义如下：

- greet-long：指定最开始的欢迎音，即最开始播放的“您好，欢迎致电某某公司，请直拨分机号，查号请拨 0”的语音，该语音文件默认的位置应该是在 /usr/local/freeswitch/sounds 目录下。应该事先把声音文件录好（可以使用 Windows 系统自带的“录音机”或其他录音软件实现，后面我们还会讲到其他录音的方法），在本例子中，我们都使用相对路径，当然也可以使用绝对路径，如 /root/ivr/welcome.wav）。另外，PSTN 交换机都是使用 PCM 编码的，如果与 PSTN 对接的话，将声音文件存储为单声道、8000Hz 的格式，能获得较好的音质并占用较少的系统资源。当然，对于其他采样率的声音文件，FreeSWITCH 也能自动转换，只是在 Log 中会有相关的采样率不匹配的警告，初学者可以暂时不必理会。
- greet-short：该项指定一个简短的提示音。如果用户长时间没有按键，则应重新提示拨号，但重新提示应该简短，比如直接说“请直拨分机号，查号请拨 0”，而不用再把公司的欢迎广告再重复播放一遍。所以，可以把这么一个声音文件录制到 welcome_short.wav 中。
- invalid-sound：如果用户按错了键，则会使用该提示。如果使用“make sounds-

install" 命令安装了声音文件，则该文件应该是默认存在的。只是它是英文的，如果需要中文的提示，可以自己录一个，并修改这里的路径使其指向自己录制的声音文件。

- exit-sound：该项指定最后菜单退出时（一般是超时没有按键）的声音，默认会提示 "Good Bye"。
- timeout：指定超时时间（毫秒），即多长时间没有收到按键就超时，播放其他提示音。
- max-failures：为容忍用户按键错误的次数。如果用户的按键与下面配置的正则表达式不匹配（即没有找到相关的菜单项），就认为是错误的。
- max-timeouts：即最大超时次数。
- inter-digit-timeout：为两次按键的最大间隔（毫秒）。如用户拨分机号 1001 时，假设拨了 10，等 3 秒，然后再按 01，这时系统实际收到的号码为 10（后面的 01 超时后没有收到），则会播放 invalid-sound 指定的声音文件以提示错误。
- digit-len：说明菜单项的长度，即最大收号位数。在本例中，用户分机号长度为 4 位，因此我们使用 4。

该菜单中有两个菜单项（Entry），第一个是在用户按 0（digits="0"）时，通过 menu-exec-app 执行一个 App。在此处它执行 transfer，将来话转到 default Dialplan 中进行路由，并会最终转到分机 1000。

如果来电用户知道被叫的分机号，则可以直接拨分机号，而不用经过人工转接，以节约时间。在本例中，第二个菜单项中的 digits 正则表达式 "/^(10[01][0-9])$/" 会匹配用户输入的 1000 ~ 1019 之间的分机，也是转到 default Dialplan 中进行路由，并最终转到对应的分机上。

如果来电用户按其他按键，则由于找不到匹配的菜单项进而提示错误（invalid-sound 指定的声音），并提示用户重新输入。

以上菜单设定好后，需要在控制台中执行 reloadxml（或按 F6 键）使配置生效。

配置完成后就可以在控制台上进行如下测试（呼叫 1001，接听后进入 ivr 菜单）：

```
freeswitch> originate user/1001 &ivr(welcome)
```

测试成功后，就可以配置 Dialplan 把用户来话转接到菜单，在 Dialplan 中加入一个 extension（注意，需要加到正确的 Dialplan Context 中，如果不确定，则应该加到哪个 Context 中，在 default 和 public 中都加上会比较保险。实际上，如果这里出了问题，应该回去看第 6 章）：

```
<extension name="incoming_call">
    <condition field="destination_number" expression="^1234$">
        <action application="answer" data=""/>
        <action application="sleep" data="1000"/>
        <action application="ivr" data="welcome"/>
    </condition>
</extension>
```

其中，正则表达式 "^1234$" 匹配 1234，即如果用户拨打 1234 时，就会在 Dialplan 中

路由到这里。其中有三个 action，分别指定接下来要执行的动作。首先，执行 answer 对来话进行应答（必须先应答才能向对方播放声音）；然后使用 sleep 暂停一秒，以防止由于声音通路没建立好（有时候，特别是当主叫与被叫之间的链路比较复杂时，声音链路的建立需要一个过程）而丢失声音；最后，执行 ivr，它的参数就是我们刚刚建立的菜单项的名字 welcome。

接下来呼叫 1234 进行测试，就可以听到我们刚才配置的 IVR 菜单了。注意，在实际应用中，为了能接受外部来的呼叫，可能要把这里的 1234 改成实际的 DID（Direct Inbound Dial）号码。

11.2.2 默认 IVR 简介

明白了以上简单的 IVR 配置方法，就很容易理解更复杂一点的配置了。上面我们提到，系统默认提供了一个名叫 demo_ivr 的菜单。最初的语音提示（greet-long 及 greet-short）是用 Phrase 实现的。Phrase 是用 XML 定义的一些短语，虽然最终也是播放相关的声音文件，但在多语言系统中会更灵活。在此我们先不讨论 Phrase，可以简单地认为它就是一个声音文件。其他各属性的含义都很直观，我们也不多介绍了。这里我们重点来看一下各菜单项。这些菜单项的配置如下：

```
    <entry action="menu-exec-app" digits="1"param="bridge sofia/$${domain}/888@
conference.freeswitch.org"/>
    <entry action="menu-exec-app" digits="2" param="transfer 9196 XML default"/>
    <entry action="menu-exec-app" digits="3" param="transfer 9664 XML default"/>
    <entry action="menu-exec-app" digits="4" param="transfer 9191 XML default"/>
    <entry action="menu-exec-app" digits="5" param="transfer 1234*256 enum"/>
    <entry action="menu-sub" digits="6" param="demo_ivr_submenu"/>
```

可以看出，菜单选项大多都是根据用户按键使用 menu-exec-app 执行相应的 App。其中，如果按 1，则会执行 bridge，连接到 FreeSWITCH 官方的电话会议平台；如果按 2，则执行 transfer，转移到 XML Dialplan 的 default Context 中的 9196 这个 extension 进行处理（对于 9196 我们已经很熟悉了，它最终会执行 echo）；如果按 3，则会转到 9664（保持音乐）；按 4 转到 9191（ClueCon）；按 5 转到 enum Dialplan 进行处理；按 6 则会使用 menu-sub 进入一个下级菜单。

menu-sub 表示会执行一个下级菜单。其实下级菜单所有的配置项目与主菜单基本上都是一样的，不同的是，在下级菜单中（此处是 demo_ivr_submenu），可以使用 menu-top 来返回上级菜单。默认的例子中下级菜单只包含一个到上级菜单的菜单项，它表示按“*”号键就可以返回主菜单。该菜单项的配置如下：

```
    <entry action="menu-top" digits="*"/>
```

使用 XML 方式配置 IVR 菜单的知识基本上只有上面介绍的这么多了。不过，通过设置多级菜单，以及与 Dialplan 相配合，根据不同的情况进行跳转，可以实现一些相对复杂的功能。当然，如果这些还不够灵活和强大，我们后面也会讲到使用嵌入式脚本或 Event Socket 实现更灵活的菜单。

如果在试验过程中遇到非预期的情况，可以打开 FreeSWITCH 控制台，按 F8 键或使用

“console loglevel debug”命令将日志调到 DEBUG 级别，在打电话时便能看到详细的执行过程。根据日志中的信息结合自己的使用经验就应该能比较容易定位到问题了。

11.3　按时间进行路由

有时候，在一些企业应用中，可能需要按时间段进行路由。例如，在上班时间路由到一个 IVR，该 IVR 在报完欢迎语后，可以引导转到人工总机接电话；而在下班后，来电就转到另外一个 IVR，工作全部由电脑自动处理。

我们可以构造如下 Dialplan：

```
<extension name="time_based_ivr">
  <condition wday="2-6" hour="8:30-17:30">
    <action application="ivr" data="ivr_day"/>
    <anti-action application="ivr" data="ivr_night"/>
  </condition>
</extension>
```

其中，我们可以看到，这里的测试条件（Condition）与以前的不同。以前我们大部分以 destination_number 为测试条件，而在这里，我们有两个测试条件，一个是 wday，一个是 hour，这两个测试条件是逻辑“与”的关系。其中，wday 表示星期（星期日的值为“0”），一般上班的时间都是周一至周五，因而这里 wday 的值用“2 ~ 6”表示；同时，这里的 hour 表示小时，即工作日的 8:30 至 17:30 为上班时间。

在有呼叫到达后，如果系统时间在该条件定义的范围内，则执行在后续的 action 中定义的 App。这里，我们只有一个 action，它只是使用 ivr App 将呼叫转入白天应该播放的 IVR（即 ivr_day）。如果定义的条件不在这个范围内，FreeSWITCH 就会执行 anti-action 指定的 App，它也是执行 ivr，不过这次它的参数是另一个 IVR——ivr_night。

当然，读到这里读者可能会有疑问，上面的例子没有测试被叫号码，会不会有什么副作用？这种担心不是多余的。这里由于没有限制被叫号码，相当于任何被叫号码都会转到 IVR，而实际上，我们只希望有人打我们的 DID 号码时才转入 IVR，所以我们的配置一般会比上面的更复杂一些。

首先，写一个 extension，它会根据不同的时间执行不同的动作，如：

```
<extension name="time_based_ivr" continue="true">
  <condition wday="2-6" hour="8:30-17:30">
    <action application="set" data="ivr=ivr_day" inline="true"/>
    <anti-action application="set" data="ivr=ivr_night" inline="true/>
  </condition>
</extension>
```

与前面的 extension 不同的是，此处的 extension 中使用了“continue=true”属性。保证在 Dialplan 解析的时候解析完此处会继续往下进行。如果当前的时间可以匹配这里的测试条件（工作日上班时间），便会执行 set，设置 ivr 变量的值为 ivr_day；否则，则将 ivr 变量的值设为 ivr_night。

注意

上面的 action 和 anti-action 都设置了 inline 属性，它的值为 true 表示该动作会立即生效，以便后续的 Dialplan 使用此处设置的 ivr 变量来进行条件判断。虽然后续我们并没有使用它来进行条件判断，但这里使用该参数仍然是一个好习惯。

总之，在路由阶段检查过这一段 Dialplan 后，变量 ivr 的值不是等于 ivr_day 就是等于 ivr_night，具体如何取决于来电的时间。另外，由于该 extension 的定义中还有“continue=true”参数，因而，Dialplan 的解析过程会继续往下执行。如果我们在它后面建立如下的 Dialplan，它将通过被叫号码匹配到来话的 DID，然后，执行 answer 进行应答，并执行 ivr 转到刚才我们设置的 ivr 变量指定的 IVR。

```
<extension name="ivr">
  <condition field="destination_number" expression="^DID$">
    <action application="answer" data=""/>
    <action application="ivr" data="${ivr}"/>
  </condition>
</extension>
```

这样，其他的被叫号码可以忽略这里的时间检查及 ivr 变量照常进行路由，不受这里的时间条件的影响。

当然，要完成完整的例子还需要配置实现 ivr_day 和 ivr_night 两个不同的 IVR，这部分内容我们在上一节刚刚讲过，读者可以自行练习。

11.4 配置中文语音提示

FreeSWITCH 中默认的提示音都是英文的，有不少中文读者都觉得不方便。几年前，在中文社区网友 link 的帮助下，笔者制作了中文的语音包，虽然现在看来有些过时，但有总比没有好。这些语音包可以在下面的地址找到：http://wiki.freeswitch.org/wiki/Language_Files#Chinese。

FreeSWITCH 对多语言的支持非常强大、灵活，下面我们先从最简单的方法学起。

11.4.1 最简单的实现方案

在 FreeSWITCH 中，默认的声音文件存放在 FreeSWITCH 安装目录的 sounds 目录下面，不同语种及嗓音的文件以不同的目录分类存放。在笔者的电脑上，用 ls 命令显示的目录结构如下：

```
dujinfang@seven:/usr/local/freeswitch/sounds/en/us/callie$ ls
ascii/       currency/   ivr/          time/
base256/     digits/     misc/         voicemail/
conference/ directory/  phonetic-ascii/ zrtp/
```

基中，en/us 表示语种，即美国英语；callie 是个人名，即这些录音是由 callie 录的。下面的子目录分别是声音文件的分类，如 ascii 代表 ASCII 字符、digits 代表数字等。当然，使

用 tree 命令可能看起来更直观一些（为了简单起见我们省略了大部分输出）：

```
dujinfang@seven:/usr/local/freeswitch/sounds/en/us/callie$ tree .
.
├── ascii
│   └── 8000
│       ├── 100.wav
│       ├── 101.wav
│       ├── 102.wav
.       .
├── digits
│   └── 8000
│       ├── 0.wav
│       ├── 1.wav
│       ├── 10.wav
│       ├── 2.wav
│       ├── 3.wav
```

可以看出，每个子目录中又都有 8000 这个子目录，它说明这些声音文件是以 8000Hz 的采样率存放的[⊖]。该目录中最后存放了实际的声音文件（读者可以自己播放来听一下）。

当然，这些文件中的语音都是英文的。我们这里所说的最简单的支持中文的方案就是将这些文件直接用中文语音包中对应的文件替换。这种方法看起来比较"土"，但很有效。

11.4.2　使用 sound_prefix

在 FreeSWITCH 的配置文件中，有一个 sound_prefix 变量用于定义声音文件的具体路径，它在 vars.xml 中是这样定义的：

```
<X-PRE-PROCESS cmd="set" data="sound_prefix=$${sounds_dir}/en/us/callie"/>
```

所以，改变该变量的值也能改变大部分声音文件的参考位置。例如，可以将其指向我们中文的语音文件路径：

```
<X-PRE-PROCESS cmd="set" data="sound_prefix=$${sounds_dir}/zh/cn/link"/>
```

当然，也可以在 Dialplan 中针对每一个 Channel 进行改变，如：

```
<action application="set" data="sound_prefix=$${sounds_dir}/zh/cn/link"/>
```

另外，该变量也可以设置到用户目录中，当特定的用户拨打电话时就能使用该变量。如，我们在 1002.xml 中可以加入以下设置：

```
<variable name="sound_prefix" value="$${sounds_dir}/zh/cn/link"/>
```

然后我们设置如下测试 Dialplan：

```
<extension name="lang">
  <condition field="destination_number" expression="^1234$">
```

⊖ 回顾一下 3.2.1 节中的 Linux 部分，上面的声音文件是我们使用"make sounds-install"命令安装的。FreeSWITCH 中支持多种频率的声音文件，如果在安装时选择了更高频率的声音文件（如使用了"make hd-sounds-install"等），则可能有 16000、32000 之类的目录。在使用时，FreeSWITCH 可以根据当前 Channel 的语音编码的采样率自动选择适合的声音文件。

```
        <action application="answer"/>
        <action application="playback" data="digits/1.wav"/>
        <action application="playback" data="digits/2.wav"/>
        <action application="playback" data="digits/3.wav"/>
    </condition>
</extension>
```

当 1002 拨打 1234 时，便能听到“一二三”，但其他用户拨打时，仍会听到英文的“one two three”。

11.4.3 使用 Phrase

当然，使用 11.4.2 节的方式更改的一般只是 playback 使用的声音文件的路径，而 FreeSWITCH 中大部分的语音提示，如 IVR 和 Voicemail 等，为了支持多语言，都是使用 Phrase 来实现的。接下来，我们就来看一下 Phrase。

1. 认识 Phrase

为了屏蔽各种不同语言提示的差异性，FreeSWITCH 实现了 Phrase（短语）框架。简单来说，通过使用它，可以将不同语言的日期、时间、货币及数字等以相同语法表示，并可以在必要时结合 TTS 实现更强大的语音提示。

在默认的 IVR 配置中，就使用了 Phrase。如在 ivr_menus/demo_ivr.xml 中，greet-long 的配置如下：

```
greet-long="phrase:demo_ivr_main_menu"
```

该 Phrase 是在 conf/lang/en/demo/demo_ivr.xml 中定义的一个宏，该宏定义的前几行代码如下：

```
<macro name="demo_ivr_main_menu" pause="100">
    <input pattern="(.*)">
        <match>
        <action function="play-file" data="ivr/ivr-welcome_to_freeswitch.wav"/>
        <action function="play-file" data="ivr/ivr-this_ivr_will_let_you_test_features.wav"/>
        <action function="play-file" data="ivr/ivr-you_may_exit_by_hanging_up.wav"/>
```

其中，name 属性定义了该宏的名称，以便在其他地方引用；pause 表示在每个 action（下面讲到）之间暂停多长时间（毫秒），它匹配（match）指定的模式（pattern）后（我们这里不讨论 match 和 pattern），便执行下面的动作（action）。本例中的动作都是一系列的 play-file，即播放后面的 data 指定的声音文件。

这些声音文件的具体路径是在 conf/lang/en/en.xml 中定义的，实际上，demo_ivr.xml 也是由该文件使用 INCLUDE 装入的。en.xml 的部分内容如下：

```
<language name="en" say-module="en"
    sound-prefix="$${sounds_dir}/en/us/callie"
    tts-engine="cepstral" tts-voice="callie">
    <phrases>
        <macros>
            <X-PRE-PROCESS cmd="include" data="demo/*.xml"/>
```

可以看出，en.xml 定义了一种语言（language），该语言的名字（name）为 en；该语言是由 say-module 指定的 en 模块（在 mod_say_en 模块中实现）支持的；所有支持它的声音文件路径都由 sound-prefix 指定；如果用到 TTS，后面还有 tts-engine 和 tts-voice 参数（我们将在 11.7 节研究 TTS）。

然后，在该语言中，定义了一些 phrase 标签，在 phrase 中又定义了一些宏（macros），具体的宏都是由后续的 XML 配置文件定义的，如上面的 IVR 中使用的宏就是在 demo_ivr.xml 中定义的。

更多关于 Phrase 的知识我们就不多介绍了，有兴趣的读者不妨拨打 9386 试一下。在 FreeSWITCH 默认的配置中拨打 9386 将播放一些有趣的声音，这些声音的定义都是在 funny_prompts 这个 Phrase 中实现的，读者可以通过这里所学的知识及跟踪 FreeSWITCH 的日志看是否能找到它们。另外，我们将在 11.8.3 节再讲一个实际的 Phrase Macro 的例子。读者也可以参考 http://wiki.freeswitch.org/wiki/Speech_Phrase_Management 进行更深入的学习。

下面我们以增加中文支持为例来看一下如何修改这些配置文件。

2. 中文 Phrase 配置

FreeSWITCH 默认没有中文的语音文件配置，因此我们需要自己添加。中文与英文的配置大同小异，在此我们将英文的复制一份，并在它的基础上进行修改。

使用如下命令复制整个目录到 zh（其中每行的第一个 # 号为系统提示符，第二个 # 为笔者加的注释）：

```
# cd conf/lang        # 进入 lang 目录
# cp -R en zh         # 从 en 目录复制产生 zh 目录
# cd zh               # 进入 zh 目录
# mv en.xml zh.xml    # 将 en.xml 改为 zh.xml
```

至此，中文配置文件的基本框架已经完成了，下面修改 zh.xml，将其中的 name 和 say-module 都修改为 zh，并把 sound-prefix 修改为中文录音文件的路径，修改后的结果如下：

```
<language name="zh" say-module="zh"
    sound-prefix="$${sounds_dir}/zh/cn/link"
    tts-engine="mod_tts_commandline" tts-voice="Ting-Ting">
```

关于 tts-engine 和 tts-voice 两个参数我们先设置到这里，等后面讲到 TTS 的时候（见 11.7 节）再详细讲解。

改好这些配置框架以后，就可以告诉 FreeSWITCH 将我们新建的中文配置文件包括进来了。在 conf/freeswitch.xml 中，可以看到如下的行：

```
<X-PRE-PROCESS cmd="include" data="lang/en/*.xml"/>
```

上面是装入英文配置文件的配置，我们只需要在该行后面加入如下的行让它也装入中文的配置：

```
<X-PRE-PROCESS cmd="include" data="lang/zh/*.xml"/>
```

重启 FreeSWITCH，中文语言的配置就基本准备好了。

3. “说”中文

另外，FreeSWITCH 中的 Say 接口（Interface）可以通过一些预先录制的声音文件“说”出一些常用的词语组合，如日期、时间、货币、数字等。它相当于一种限制级的 TTS 实现（之所以说是限制级，就是因为它虽然非常灵活，但只能读特定范围内的词）。

Say 接口支持各种语言，如果需要支持中文语言，就需要 mod_say_zh 模块。该模块默认是不被编译也不被加载的，因此我们需要先编译它。到 FreeSWITCH 的源代码目录中，执行如下命令即可编译安装：

```
make mod_say_zh-install
```

然后在 FreeSWITCH 控制台上加载该模块：

```
freeswitch> load mod_say_zh
```

为了让 FreeSWITCH 在启动时能自动加载该模块，需要在 conf/autoload_configs/modules 中将下列行的注释去掉，以让 FreeSWITCH 自动加载该模块。去掉注释后的结果为：

```
<load module="mod_say_zh"/>
```

关于 Say 的例子我们将在后面讲到，暂时先把它准备好，后面的例子会用到。

11.4.4 使用中文语音提示

中文支持准备就绪后，为了能播放中文提示，还需要在 Dialplan 中指定 language 或（和）default_language 通道变量[⊖]。比如，我们修改默认的 5000 对应的 Dialplan，加入以下配置：

```
<action application="set" data="language=zh"/>
```

完整的 Dialplan 如下：

```
<extension name="ivr_demo">
    <condition field="destination_number" expression="^5000$">
        <action application="set" data="language=zh"/>
        <action application="answer"/>
        <action application="sleep" data="2000"/>
        <action application="ivr" data="demo_ivr"/>
    </condition>
</extension>
```

另外，该变量也可以加到用户目录中。比如，我们可以在 1002 的用户配置文件 1002.xml 的 variables 标签中添加如下配置，让该语言仅对 1002 用户生效：

```
<variable name="language" value="zh"/>
<variable name="default_language" value="zh"/>
```

也可以将上述配置添加到 conf/directory/default.xml 中（也就是添加到 variables 标签中），以使该配置对本域中的所有用户都生效。

⊖ 关于 language 和 default_language，有的情况需要前者，有的情况需要后者，所以，为了保险起见，我们最好两者都配置。

总之，不管使用哪种配置，拿起对应的分机拨打 5000，就应该能听到中文的语音提示了。

此话题我们就先讲到这里，有条件的读者不妨也试着同步配置一下。在本章后面的部分，我们将看到更多使用中文的语音提示的实例。

11.5 录音

在实际使用过程中，经常需要对通话进行录音。FreeSWITCH 支持多种录音方式，也支持多种格式的录音文件。在此，我们逐一尝试一下。

11.5.1 单腿录音

如在上一节提到的，在创建 IVR 或比较复杂的呼叫流程时，我们经常需要一些录音文件，除了使用 Windows 录音机或其他软件外，我们还可以直接通过 FreeSWITCH 来实现。

笔者经常使用以下命令录音，这种方法非常快速和方便：

```
originate user/1000 &record(/tmp/welcome.wav)
```

上述命令大家已经很熟悉了，我们呼叫 1000，1000 接听后即可以直接讲话并录音。由于这种录音方式仅涉及一条腿（leg，即一个 Channel），因而称为单腿录音。

另外，我们也可以在 Dialplan 中录音（与上述命令是等价的）：

```
<extension name="record">
    <condition field="destination_number" expression="^rec(.*)$">
        <action application="answer"/>
        <action application="playback" data="tone_stream://%(100,1000,800)"/>
        <action application="record" data="/tmp/$1.wav"/>
    </condition>
</extension>
```

通过设置上述 Dialplan，呼叫一个 rec 开头的号码，如 recwelcome，就可以开始录音并将录音文件保存到 /tmp/welcome.wav 中，同样也可以通过拨打其他的号码录不同名称的声音文件。另外，为了在录音前给个提示，可以在 record 前先用 playback 播放一段提示音，如“请在嘀声后开始录音”。在本例中，我们仅用 playback 播放了一个“嘀”声，该声音是用 tone_stream 产生的。我们后面还会讲到 tone_stream。

笔者使用的软电话可以呼叫含有字母的被叫号码。如果不支持呼叫含有字母的号码，也可以把这里的 rec 换成其他的数字，如 123。然后，呼叫 1231000 就可以将录音存到 /tmp/1000.wav。

在这种录音方式中，由于只有一个 Channel，所以录音文件是单声道的。我们可以直接用下面的方式使用 playback 来测试播放刚才的录音：

```
freeswitch> originate user/1000 &playback(/tmp/welcome.wav)
```

上述命令呼叫 1000，1000 接听后就可以听到刚刚录制的声音了。

11.5.2 对两条腿的通话进行录音

一路正常的通话通常由两个 Channel 组成。而对于每个 Channel 我们都可以单独录音。在一个 Channel 中，语音有两个方向。对于 SIP 客户端或话机而言，两个方向分别为“说”和“听”，而对于 FreeSWITCH 而言，则分别为“读”(read，r) 和“写”(write，w)。

以下 API 命令可以对正在进行的通话录音（需要事先知道相关 Channel 的 UUID）：

```
uuid_record <channel_uuid> start /tmp/record.wav
```

如果决定不录音了，则可以使用以下命令停止录音：

```
uuid_record <channeL_uuid> stop /tmp/record.wav
```

该录音文件会包含两个声道，以后要使用 playback 播放这种录音时，会在日志中看到如下的警告信息：

```
freeswitch> originate user/1000 &playback(/tmp/record.wav)
[WARNING] switch_core_file.c:229 File has 2 channels, muxing to mono will occur.
```

这条信息表示，该文件有两个声道，但 playback 仅是对一个 Channel 而言的，它仅支持一个声道。因而，FreeSWITCH 会先将两个声道混音，变成一个声道后再播放。该警告一般是无害的。只是，由于在播放时会进行混音，会多占用一些 CPU，因而在高并发的场合，可以使用一些工具事先将声音文件混为一个声道。如我们可以使用 sox 命令行进行混音：

```
$ sox record.wav -c 1 record-1.wav
```

其中，“ -c”指定 Channel 的数量。我们这里用的是“ -c 1”，就是说将原来的声音文件混成一个声道。可以使用以下 ls 命令，从中看到混音后的声音文件大小比原来的大约小了一半（从 136364 到了 68204）：

```
$ ls -l /tmp/record*.wav
-rw-r--r--  1 dujinfang  wheel  136364 Sep 23 15:21 record.wav
-rw-r--r--  1 dujinfang  wheel   68204 Sep 23 15:45 record-1.wav
```

另外，也可以使用 sox 附带的 play 命令在操作系统命令行上播放声音文件，并查看一些详细的信息，如（注意，该命令在声卡驱动不正常时是不能使用的）：

```
$ play /tmp/record.wav

/tmp/record.wav:

 File Size: 136k      Bit Rate: 256k
  Encoding: Signed PCM
  Channels: 2 @ 16-bit
Samplerate: 8000Hz
Replaygain: off
  Duration: 00:00:04.26
```

可以看到，上述输出中声音编码（Encoding）是有符号的（Signed）PCM 数据；有两个 16bit 的声道（Channel)；采样率（Samplerate）是 8000Hz，文件的长度是 4.26 秒。读者也可以对比一下它与上面单声道录音文件的区别。

实际上，录音使用的是 FreeSWITCH 中的 Media Bug 功能（即 8.3.4 节讲的“三通”），声音媒体数据通过一个“三通”流到录音文件中。FreeSWITCH 支持同时创建多个 Media Bug，因而可以同时录制多个声音文件，这在需要截取不同时间段的录音时非常有用，如可以在不同的时间点分别执行下列命令：

```
uuid_record <channel_uuid> start /tmp/1.wav
uuid_record <channel_uuid> start /tmp/2.wav
uuid_record <channel_uuid> start /tmp/3.wav
```

可以随时用对应的 stop 指令停止某个录音，也可以使用特殊的文件名“ all”同时停止所有录音，如：

```
uuid_record <channel_uuid> stop all
```

当然，如果在录音过程中电话挂断了，所有录音也就自动停止了。

另外，在 Dialplan 中，也可以通过 record_session 达到类似的效果，如：

```
<extension name="record">
    <condition field="destination_number" expression="^(100[0-9])$">
        <action application="record_session" data="/tmp/record-$1.wav"/>
        <action application="bridge" data="user/$1"/>
    </condition>
</extension>
```

其中，在执行 bridge 前使用 record_session 开始对当前的 Channel 进行录音。与 uuid_record 不同的是，record_sesison 是一个 App，而前者是一个 API。相同的是两者在执行时都会为当前的 Channel 添加一个 Media Bug，因而可以实时录音。

另外，与 record 比起来，record_session 是非阻塞的，它可以作用于单腿的呼叫也可以用于桥接（Bridge）的呼叫；而 record 是阻塞的，它会阻塞直到录音完成，即只能录单腿的呼叫。

11.5.3　立体声

通过使用相关参数，可以录制立体声，即可以将通话的两个人的声音分别存在两个声道里。该特性非常有用。笔者最早在做网络远程一对一教学时，就曾使用了该技术。通过将通话中的老师和学生的声音录在不同的声道里，可以在事后很方便地检查出他们对话的内容，并有针对性地提出建议以提高教学质量。另外，我们也可以很方便地将不同的声道分离到不同的声音文件中，进行更高级的自动声音识别和分析。

通过事先将 RECORD_STEREO 通道变量设置为 true 可以在录音时直接录成立体声。在 Dialplan 中使用 record_session 时的设置如下：

```
<action application="set" data="RECORD_STEREO=true"/>
<action application="record_session" data="/tmp/record-$1.wav"/>
```

当然，也可以在使用 uuid_record 录音前使用 uuid_setvar 设定该通道变量，从而录制成立体声的声音文件。这种方式与上述方式等价，具体命令如下：

```
freeswitch> uuid_setvar <channel_uuid> RECORD_STEREO true
freeswitch> uuid_record <channel_uuid> start /tmp/record.wav
```

11.5.4 录音相关的通道变量

除了上面讲到的 RECORD_STEREO 外，还有一些与录音相关的通道变量，下面简单介绍一下（注意这些变量是区分大小写的）：

- RECORD_HANGUP_ON_ERROR：默认值为 false，如果设置为 true 则会在录音失败时（如文件不可写入、格式错误等）挂断电话。
- RECORD_WRITE_ONLY：只录“写”方向的录音。这里的方向是相对 FreeSWITCH 而言的，即 FreeSWITCH 发出，也就是对端能够听到的声音。
- RECORD_READ_ONLY：只录“读”方向的录音，即 FreeSWITCH“听”到的录音。
- RECORD_STEREO：立体声录音。
- RECORD_STEREO_SWAP：同立体声录音，但将左、右声道互换。
- RECORD_ANSWER_REQ：默认值为 false，如果设为 true，则在应答前执行了 record_session 或 uuid_record 就暂时不录音，而等到通话被应答后才开始录音。可以防止录上不必要的 Early Media（如回铃音或彩铃等）。
- RECORD_BRIDGE_REQ：与 RECORD_ANSWER_REQ 类似，只是在当前的通道与其他通道桥接了之后才开始录音。
- RECORD_APPEND：在默认情况下，如果指定的录音文件已经存在，再次录音就会覆盖该文件。将该选项设为 true 可以向已经存在的文件后面追加录音。可用于电话中断重新建立后重新录音时。
- record_sample_rate：可以在录音时指定采样率，以进行实时的转码。合法的采样率在源代码中使用 switch_is_valid_rate(rate) 宏定义，可能的取值有 8000、12000、16000、24000、32000、11025、22050、44100、48000，单位为 Hz。
- enable_file_write_buffering：该变量的默认值为 true，即默认先写到一个内存缓冲区中，可以大大减小对存储设备（如硬盘）的访问次数，防止产生 IO 瓶颈。默认的缓冲区大小是 65536 字节。
- RECORD_MIN_SEC：最小录音秒数。如果录音时长小于该数（默认是 3 秒），则认为是无效的录音，并删除录音文件。
- RECORD_INITIAL_TIMEOUT_MS：如果从录音开始检测不到声音，则在指定的毫秒后超时，停止录音。
- RECORD_FINAL_TIMEOUT_MS：如果在录音过程中检测不到声音，则在指定的毫秒后超时，停止录音。
- RECORD_SILENCE_THRESHOLD：静音能量阈值，即能量小于该值认为是静音。默认值为 200。
- 其他：还有一些变量可以设置声音文件的元信息，名字都很直观，在此就不多介绍了。这些变量有 RECORD_TITLE、RECORD_COPYRIGHT、RECORD_SOFTWARE、RECORD_ARTIST、RECORD_COMMENT、RECORD_DATE 等。

11.5.5 原生格式

为了最大限度地节省系统资源，可以将声音录制成原生（Native）格式[⊖]。这种格式是由 mod_native_file 模块提供的。如果在录音时不提供录音文件的扩展名，就可以以原生的格式录音，例如，下列命令会将录音文件录成 /tmp/test.PCMU（假设 Channel 使用的 Codec 是 PCMU）：

```
freeswitch> originate user/1000 &record(/tmp/test)
```

对于一个两条腿的通话，也可以使用 uuid_record 进行原生方式的录音，如：

```
freeswitch> uuid_record b40dec66-2f47-480d-88bb-335a025d856e start /tmp/test
[INFO] mod_native_file.c:94 Opening File [/tmp/test-in.PCMU] 8000hz
[INFO] mod_native_file.c:94 Opening File [/tmp/test-out.PCMU] 8000hz
[DEBUG] switch_core_media_bug.c:532 Attaching BUG to sofia/internal/
1001@123.130.116.42
```

从日志中可以看出，由于该通话有两条腿，而原生的格式不支持混音，因此只能将双向的声道录到两个不同的文件中，如上面的“-in”和“-out”分别代表“入”和“出”。

使用这种录音方式，一个最大的好处是可以直接支持 G729 编码。G729 是受专利保护的编解码器，因而 FreeSWITCH 中不包含开源的编解码实现。通过使用原生格式录音，不需要进行编码和解码，因而就可以绕开这种限制。另外，使用这种方式也不用在 FreeSWITCH 中进行编解码处理，因而也会减少对系统 CPU 的使用。

如果 Channel 使用 G729 编码，则原生格式的录音文件扩展名将是 .G729。这种格式的文件可以再使用其他的 G729 工具进行处理，我们就不多介绍了。在本节结束之前，我们再来讲一下 PCM 格式的文件的处理。

PCM 格式的原生文件可以使用 sox 软件中的 play 命令播放，如：

```
play -e u-law -r 8000 -t raw test-in.PCMU
```

注意，由于原生格式的录音文件中不包含录音数据的元信息（即编码、采样率、声道等参数），因而需要在播放时指定。在上述命令中，“-e”指定了编码为“u-law”，即 μ 律；“-r”参数指定采样率为 8000Hz；-t raw 说明以原生格式播放。

当然，也可以使用如下的 sox 命令将该原生格式的文件转换成 WAV 文件：

```
sox -e u-law -r 8000 -t raw test-in.PCMU test-in.wav
sox -e u-law -r 8000 -t raw test-out.PCMU test-out.wav
```

也可以进一步将上述两路文件合成一个文件。下列两条命令都可以将两个声音文件合成到一个中去，其中前者将进行混音，而后者将合成立体声的。

```
sox test-in.wav test-out.wav -m test.wav
sox test-in.wav test-out.wav -M test.wav
```

更多的使用方法请参阅有关资料。

⊖ 即直接使用音频媒体数据的原始格式（如 PCMU、G729 等），而不需要转换成其他编码。

11.6 放音

上一节我们讲了各种录音方式的参数。实际上，放音也有好多门道。接下来我们来详细看一下。

11.6.1 playback 的参数

我们先回顾一下，在前面的章节中，我们一直使用 playback 作为放音的例子。下列 Dialplan 设置可以播放声音文件 /tmp/test.wav：

```
<action application="playback" data="/tmp/test.wav"/>
```

其中，/tmp/test.wav 是一个文件名作为 playback 的参数。实际上，playback 的参数范围比这个广得多，最典型的，playback 的参数是一些“音频源”（以后，也可能有视频源），这些音频源大部分是由 Format 定义的，即我们在 5.1.2 节讲到的格式、文件接口（Format，File Interface）。这些文件接口一般会有打开（Open）和关闭（Close）、读（Read）和写（Write）等属性。其中，playback 会从文件中“读”，而上面讲到的录音（record），实际上也使用这些文件接口，它会往文件中“写”。

这些文件接口有以下几类。

1. 声音文件

对大部分声音文件的支持都是在 mod_sndfile 模块中实现的。该模块直接调用了 libsndfile 库，因而 libsndfile 库所支持的文件 FreeSWITCH 都能支持，典型的如 WAV、AU、AIFF、VOX 等。

```
<action application="playback" data="/tmp/test.wav"/>
<action application="playback" data="/tmp/test.au"/>
<action application="playback" data="/tmp/test.aiff"/>
```

libsndfile 不支持的声音文件则由其他模块实现的。如 mod_shout 模块实现了对 MP3 文件的支持。该模块默认是不编译的，可以在源代码目录中使用如下命令编译安装：

```
# make mod_shout-install
```

然后，在 FreesSWITCH 中使用下列命令加载该模块：

```
freeswitch> load mod_shout
```

然后就可以播放 MP3 文件了：

```
<action application="playback" data="/tmp/test.mp3"/>
```

当然，该模块也支持播放远程 SHOUTcast 服务器上的 MP3 文件，如：

```
<action application="playback" data="shout://shoutcase-server/test.wav"/>
```

另外，Playback 也可以直接播放由 mod_native_file 模块（C 参见 11.5.5 节）支持的原生文件——只要在播放时不带扩展名。FreeSWITCH 会自己查找与本 Channel 语音编码一致的文件。在以下设置中，如果 Channel 使用 PCMU 编码，则播放 test.PCMU，如果 Channel 使

用 G729 编码，则会播放 test.G729：

```
<action application="playback" data="/tmp/test"/>
```

在 FreeSWITCH 控制台上使用“show file”命令可以列出哪些模块都实现了哪些文件类型的支持，读者可以自己试验一下，在这里就不多讲了。

2. local_stream

local_stream 是在 mod_local_stream[⊖]中实现的。该模块实现了一些 Stream，即“流”。它与文件类似，不同的是，每个流在整个系统中只有一个实例，但可以同时被多个 Channel 读取。这样，当系统中有成千上万个 Channel 时，便能节省很多系统资源。由于流的这个特性，使得它非常适合做保持音乐（Music on Hold，MOH）。

比如，我们熟悉的 FreeSWITCH 默认设置中的 9664 号码就是用下列配置播放了一个流：

```
<action application="playback" data="local_stram://moh"/>
```

上述代码中的流的名字是“local_stream://moh”，它是在 conf/autoload_configs/local_stream.conf.xml 中定义的。我们可以在配置文件中找到如下的配置：

```
<directory name="moh/8000" path="$${sounds_dir}/music/8000">
    <param name="rate" value="8000"/>
    <param name="shuffle" value="true"/>
    <param name="channels" value="1"/>
    <param name="interval" value="20"/>
    <param name="timer-name" value="soft"/>
</directory>
```

可以看出，上述代码中的流的名字是 moh/8000，FreeSWITCH 支持多种采样频率，如果当前 Channel 为 8000Hz，名字为 local_stream://moh 便可以自动对应到 moh/8000 这个流。同理，如果是 16000Hz 的 Channel，就会找到 moh/16000 定义的流。

其后面的 path 定义了该流所需的声音文件的路径，它指定一个目录，在 mod_local_stream 模块加载时会预先加载所有声音文件。有时候，如果在最初安装 FreeSWITCH 时忘了安装声音文件，该流便不能生效，在使用时会遇到类似如下的错误：

```
[WARNING] mod_local_stream.c:469 Unknown source moh, trying 'default'
[ERR] mod_local_stream.c:478 Unknown source default
```

在 FreeSWITCH 中使用流时，如果找不到指定的流，则会尝试使用 default 流代替；如果 default 流也不存在，则会显示“Unknown source default”错误，即在前面看到的出错信息。

出现上述错误一般是在安装 FreeSWITCH 时忘了安装相关的声音文件，可以尝试到源代码目录中执行如下命令以安装默认的声音文件：

```
# make moh-install
```

然后，到 FreeSWITCH 控制台上执行下列命令重新加载模块使之生效：

⊖ 参见：http://wiki.freeswitch.org/wiki/Mod_local_stream。

```
freeswitch> reload mod_local_stream
```

3. silence_stream

silence_stream 是一个静音流，它是在 mod_tone_stream 中实现的。使用方法如下：

```
<action application="playback" data="silence_stream://2000"/>
```

其中，2000 是指定播放的时间，单位是毫秒，即上述配置会播放 2 秒的静音。

在 Dialplan 中，有一个 sleep App 在某种程度上可以达到与 playback 同样的效果。如下列命令会使当前 Channel 的执行暂停 2 秒，表现在对端就是 2 秒内听不到任何声音：

```
<action application="sleep" data="2"/>
```

但两者还是有区别的。slience_stream 会播放静音，也就是说，仍然会有 RTP 数据发送，只是 RTP 中携带的声音数据中音量全部为 0；而 sleep 会中断 RTP 流的传输。有时候，中断 RTP 流的传输后可能会造成对方功能失常，在这种情况下，就可以播放 slience_stream 以代替 sleep。

如果在 silence_stream 中指定时长为 0，则默认为无限长，如下列配置会一直播放静音：

```
<action application="playback" data="silence_stream://0"/>
```

使用上面介绍的 silence_stream 播放的静音称为绝对静音，因为它产生的静音音量绝对为 0。但有时候，在电话中听起来，绝对静音与电话断了没什么区别，因而电话的另一端的人有时会误以为电话断了，进而可能会错误地挂断电话。为避免这种情况，最好能像普通电话一样能产生轻微的“嘶嘶”声，这样对方就知道电话还是通的。这种“嘶嘶”声音量不大，恰好能被听到，又不招人讨厌，一般被称为“舒适噪声”(Comfortable Noice，CN)。

可以在 silence_stream 中使用逗号加上第二个参数让它产生舒适噪声，如下面的配置将产生时长为 2 秒的舒适噪声：

```
<action application="playback" data="silence_stream://2000,1400"/>
```

其中，1400 为舒适噪声的参数值，该参数的取值范围是 1 ~ 10000，值越小噪声的音量越大，通常使用的参数值为 600 ~ 1400。在不同的场景下具体哪个值合适，读者可以自己试一下。另外，如果将该参数值设为“-1”，则它会产生绝对静音。

4. tone_stream

tone_stream 是一个铃流，它是在 mod_tone_stream ⊖模块中实现的。它可以使用 TGML 语言生成各种信号音。如我国的回铃音信号为 450Hz 的信号音，通断时间为 1 秒通，4 秒断。所以，可以用如下参数产生回铃音信号（其中，1000、4000 单位均为毫秒）：

```
<action application="playback" data="tone_stream://%(1000,4000,450)"/>
```

美国的回铃音信号是 440Hz 和 480Hz 交替的，通断时间分别为 2000 毫秒和 4000 毫秒：

```
<action application="playback" data="tone_stream://%(2000,4000,440,480)"/>
```

⊖ 参见：http://wiki.freeswitch.org/wiki/Mod_tone_stream。

而英国的回铃音更复杂一点，读者可以思考一下它的含义：

```
<action application="playback"data="tone_stream://%(400,200,400,450);%(400,2000,400,450)"/>
```

conf/vars.xml 中还定义了一些其他国家的信号音。

除此之外，也可以在后面再附加别的参数，如 loops=3 表示循环产生 3 次信号音：

```
<action application="playback" data="tone_stream://%(1000,4000,450);loops=3"/>
```

如果 loops=-1，则表示无限循环。更多的参数我们就不多介绍了，读者可以参考 http://wiki.freeswitch.org/wiki/TGML 进行进一步的学习。

5. file_string

file_string 文件接口原先是在 mod_file_string[⊖]模块中实现的，后来被合并到了 mod_dptools 中。它相当于一种更高级的文件格式。通过它，可以将多个文件串联起来，放到同一个 playback 命令中播放，如下列配置将会顺序播放三个文件：

```
<action application="playback"data="file_string:///tmp/file1.wav!/tmp/file2.wav!/
tmp/file3.wav"/>
```

其中，“!”为多个文件之间的分隔符。如果文件名中含有该符号，也可以通过 playback_delimiter 通道变量来改变它的默认值。例如，在下列配置中将多个文件的分隔符变成“|”(注意，这时“!”就可以作为文件名的一部分，如 file3!.wav)：

```
<action application="set" data="playback_delimiter=|"/>
<action application="playback"
    data="file_string:///tmp/file1.wav|/tmp/file2.wav|/tmp/file3!.wav"/>
```

另外，也可以指定多个文件间的播放时间间隔，例如下列设置将时间间隔设为 500 毫秒：

```
<action application="set" data="playback_sleep_val=500"/>
<action application="playback" data="file_string://file1.wav!file2.wav"/>
```

6. 其他接口与参数

mod_vlc 模块实现了一个 vlc 的文件接口。VLC 是一个跨平台的多媒体播放器，它能播放主流音、视频文件。而 libvlc 则是从 VLC 中独立出来的一个库，mod_vlc 便使用了该库支持各种 libvlc 能支持的文件类型。如下列配置可以使用 mod_vlc 中的 vlc 文件接口播放一个本地文件：

```
<action application="playback" data="vlc:///tmp/test.mp3"/>
```

也可以播放视频文件中的音频部分：

```
<action application="playback" data="vlc:///tmp/test.mp4"/>
```

也可以播放远程 HTTP 服务器上的文件：

```
<action application="playback" data="vlc://http://192.168.0.2/test.mp3"/>
```

⊖ http://wiki.freeswitch.org/wiki/Mod_file_string。

当然，另外有一个 mod_httapi 模块直接实现了一个 HTTP 接口（可能需要事先编译加载），与 vlc 接口不同的是，它可以将远程文件缓存到本地，使用方法如下：

```
<action application="playback" data="http://192.168.0.2/test.mp3"/>
```

除了各种文件接口外，在 playback 的参数中添加“say:”前缀也可以直接调用 TTS 功能（关于 TTS 的功能，我们在 11.7 节会讲到），如：

```
<action application="playback" data="say:tts_commandline:Ting-Ting:欢迎使用 Free-SWITCH"/>
```

或（下列配置与上一行是等价的）：

```
<action application="set" data="tts_engine=tts_commandline"/>
<action application="set" data="tts_voice=Ting-Ting"/>
<action application="playback" data="say:欢迎阅读 FreeSWITCH 权威指南"/>
```

当然，上述的 playback 与下面的 speak 也是等价的，如：

```
<action application="speak" data="tts_commandline|Ting-Ting|欢迎阅读 FreeSWITCH 权威指南"/>
```

另外，也可以在 playback 中使用“phrase:”前缀播放 Phrase 宏，如：

```
<action application="playback" data="phrase:demo_ivr_main_menu"/>
```

它等价于：

```
<action application="phrase" data="demo_ivr_main_menu"/>
```

那么，既然有等价的 speak 和 phrase App，为什么要在 playback 中实现这种带前缀的格式呢？答案是：它有助于用在其他类似 playback 的地方，如它可以用于 IVR 的配置中，默认 IVR 的配置中的欢迎音就是用带有 phrase: 前缀的 Phrase 宏实现的。参考配置如下：

```
greet-long="phrase:demo_ivr_main_menu"
```

另外，这种格式也可以直接用于 read 或 play_and_get_digits App 中需要一个声音文件参数的地方，如：

```
<action application="set"  data="tts_engine=tts_commandline"/>
<action application="set"  data="tts_voice=Ting-Ting"/>
<action application="read" data="1 1 say:请您选择提示语言，1 为普通话，2 为英语"/>
```

关于 playback 的参数的例子我们就讲到这里，读者可以自己多多练习，找到最适合自己的方法。

11.6.2 循环播放

有些文件接口类型本身就支持循环播放，如各种 Stream 的实现有的天生就是循环的，有的可以用参数控制实现循环。而对于单纯的声音文件，则一般无法实现循环，如果要多次播放，则可以多次调用 playback，或使用 file_string 实现。

另外，系统也提供了 endless_playback 和 loop_playback 两个 App 用于多次播放某个文件。顾名思义，前者会无限循环地播放一个声音文件，后者会播放一个声音文件并重复播放

指定的次数。比如，不断播放 test.wav 直至挂机的实现如下：

```
<action application="endless_playback" data="/tmp/test.wav"/>
```

循环播放 test.wav 文件 3 次后停止播放的命令如下：

```
<action application="loop_playback" data="+3 /tmp/test.wav"/>
```

11.6.3　Say

除了 playback 外，FreeSWITCH 中还有其他放音的方法。本节要讲的 Say 是一个接口（Interface），它通过定义一个统一的接口屏蔽了语言的多样性。Say 也是一个 App，它可以通过特定的语法，使用预先录好的声音读出日期、时间等，并通过 Say 接口支持多语言（语种）。我们先来看一个例子。下面的例子将不使用 TTS，而直接使用系统预先录好的文件读出 "One Two Three Four"（注意，这里我们选的是英语的发音）：

```
<action application="say" data="en number iterated 1234"/>
```

其中，参数 en 代表语种，这里是英语；number 说明我们要播放的数据的类型，这里表示的是数字；iterated 是指定播放的方式，这里说明读数字要逐个读出；最后的 1234 即实际要读的内容。

实际上，Say 会根据一定的规则找到 1.wav、2.wav、3.wav、4.wav 并播放出来。

Say 接口能读的数据类型如表 11-1 所示，但注意并不是所有语言都实现了所有类型的读法。

表 11-1　Say 能读的数据类型

内　容	说　明
NUMBER	数字
ITEMS	项目
PERSONS	人名
MESSAGES	消息
CURRENCY	货币
TIME_MEASUREMENT	时间
CURRENT_DATE	当前日期
CURRENT_TIME	当前时间
CURRENT_DATE_TIME	当前日期和时间
TELEPHONE_NUMBER	电话号码
TELEPHONE_EXTENSION	分机号
URL	网址
IP_ADDRESS	IP 地址
EMAIL_ADDRESS	Email 地址
POSTAL_ADDRESS	邮寄地址
ACCOUNT_NUMBER	账号

（续）

内 容	说 明
NAME_SPELLED	姓名
NAME_PHONETIC	姓名拼写
SHORT_DATE_TIME	短日期时间

在确定了数据类型后，具体的读法有以下三种（以数字 27 为例）：

- PRONOUNCED：按数字语音语出，如英文“Twenty seven”，中文“二十七”；
- ITERATED：逐个读出，如英文“Two, Seven”，中文“二、七”；
- COUNTED：以序数词方式读出，如英文“Twenty senventh”，中文“第二十七”。

了解了这些以后，我们来看下面几个例子：

```
<action application="say" data="en NUMBER ITERATED 1234"/>
<action application="say" data="en NUMBER PRONOUNCED 1234"/>
<action application="say" data="en CURRENCY PRONOUNCED 1234.56"/>
<action application="say" data="en CURRENT_DATE PRONOUNCED ${strepoch()}"/>
```

由于上面这一组使用的是英文，因此电话路由到这里后将依次读出：

- One two three four
- One thousand two hundred thirty-four
- One thousand two hundred thirty-four dollars and fifty-six cents
- October nine, two thousand thirteen

其中 ${strepoch()} 将自动计算当前时间[⊖]，所以这里最后一项日期读出的结果取决于当前的时间。

我们接着来看下一组：

```
<action application="say" data="zh NUMBER ITERATED 1234"/>
<action application="say" data="zh NUMBER PRONOUNCED 1234"/>
<action application="say" data="zh CURRENCY PRONOUNCED 1234.56"/>
<action application="say" data="zh CURRENT_DATE PRONOUNCED ${strepoch()}"/>
```

这一组使用中文，将依次读出：

- 一二三四
- 一千二百三十四
- 一千二百三十四元五十六分
- 二零一三年十月九日星期三

对比一下可以看出，中文与英文除了日期中多了个星期三外，还有一个很特别的地方，就是货币的读法是非常不符合中文的习惯。

就这个问题，笔者曾与原模块的作者交流过。他说如果改成我们的习惯以后，可能会破坏其他地区中文的朗读习惯。笔者想了想，他说的确实有道理。比方说，在美国的人，即使

⊖ strepoch 是一个 API 命令，它会输出当前的 UNIX 时间戳，比如，作者写本节时该命令的输出结果是 1381332083。

说中文，但他们指的货币也是美元，可能仍会按照美元的习惯读“五十六分”而不是读“五毛六”。

后来，笔者就又依照 zh 这个接口，增加了 zh_CN 接口，让它符合我们中国人的习惯。具体的实现代码及补丁我们后面会讲到。在这里，如果要实现配置和使用这个接口，只需要仿照我们在 11.4.3 节讲的将 conf/lang/zh/zh.xml 文件复制一份到 conf/lang/zh/zh_CN.xml，并将文件中 language 标签的 name 属性改成“zh_CN”就可以了，如：

```
<language name="zh_CN" ...
```

然后，重新启动 FreeSWITCH，下面的 Say 就可以说地道的中文了。如下面的配置读出的结果是“一千二百三十四元五角六分”。

```
<action application="say" data="zh_CN CURRENCY PRONOUNCED 1234.56"/>
```

当然，为了“好玩”，笔者还增加了“厘”的读音。如下面的配置将会读出“一千二百三十四元五角六分七厘八”：

```
<action application="say" data="zh_CN CURRENCY PRONOUNCED 1234.5678"/>
```

当然如果小数点后有更多的位数，也可以原样读出。不过好玩归好玩，它的“忍耐力”也是有限的，实际使用时应该事先将数字截取到合情合理的精度。

最后，我们再讲一个例子。有的读者可能想，能否让它将“1234.56”读成“一千二百三十四点五六元”呢？据笔者所知，目前现成实现是没有的，而且在 NUMBER 的读法中也没有读小数的功能。不过，我们可以变通一下，用下列配置结合 playback 达到我们的目的。这样做虽然不是很完美，但总算能达到我们要的效果。具体的原理读者可以自己思考一下，在此就不多讲了。

```
<action application="say" data="zh NUMBER PRONOUNCED 1234"/>
<action application="playback" data="digits/point.wav"/>
<action application="say" data="zh NUMBER ITERATED 56"/>
<action application="playback" data="currency/yuan.wav"/>
```

11.7 TTS

TTS（Text To Speech）是将文本转换成语音的一项技术，因而又称为语音合成（Synthesis）。下面我们来看一下 FreeSWITCH 中 TTS 的用法。

11.7.1 使用 mod_flite

mod_flite [1]是 FreeSWITCH 基于 Flite [2]语音合成引擎的一个 TTS 模块。目前，该模块

[1] 参见 http://wiki.freeswitch.org/wiki/Mod_flite。

[2] Flite 是 Festival-Lite 的缩写。Festival 是爱丁堡大学研究出的一款开源语音合成引擎，致力于支持多语言及多平台。而 Flite 相当于 Festival 的简易版，致力于运行在小的嵌入式系统上。详见 http://www.festvox.org/flite/。

仅支持英文。不过，我们可以先拿它来做个实验，学习一下 FreeSWITCH 中 TTS 的用法。

首先，到 FreeSWITCH 源代码目录中使用下列命令编译安装 TTS 模块：

```
# make mod_flite-install
```

然后，到 FreeSWITCH 中加载 TTS 模块：

```
freeswitch> load mod_flite
```

加载 TTS 模块后，就可以使用 speak App 来调用它了。我们先来试一下以下命令：

```
originate user/1000 &speak('flite|kal|Hello, Welcome to FreeSWITCH')
```

上述命令呼叫 1000，并执行 speak。其中 speak 的参数是使用竖线隔开的。第一个参数 flite 表示 TTS 引擎的名字；第二个参数 kal 表示一种嗓音（Voice），一般是一个人名。mod_flite 目前支持 awb、kal、rms、slt 四种嗓音。执行上述命令，在分机 1000 振铃并接听后，将可以听到 “Hello, Welcome to FreeSWITCH”。

我们也可以做以下 Dialplan 进行上述测试：

```
<extension name="TTS">
    <condition field="destination_number" expression="^1234$">
        <action application="answer"/>
        <action application="speak" data="flite|rms|Hello, Welcome to FreeSWITCH"/>
    </condition>
</extension>
```

配置好上述 Dialplan 后，拨打 1234，也能听到同样的声音。

除了在 speak 的参数中指定 TTS 引擎和嗓音参数外，也可以通过通道变量指定。如下列配置与上述 Dialplan 是等价的（我们在 11.6.1 节也讲过这种等待关系）：

```
<action application="set" data="tts_engine=flite"/>
<action application="set" data="tts_voice=kal"/>
<action application="speak" data="Hello, Welcome to FreeSWITCH"/>
```

11.7.2 mod_tts_commandline

许多 TTS 软件都有命令行版本，可以使用命令行来执行 TTS 功能。mod_tts_commandline 模块可以调用这些命令，生成一个声音文件，进而播放这个声音文件，从而达到利用其他 TTS 软件执行 TTS 功能的目的。

1. 使用 Mac 上的 TTS 功能

在笔者使用的 Mac 系统上，有一个内置的 say 命令可以播放 TTS，如在命令行上执行如下命令便可以听到声音：

```
$ say Hello, Welcom to FreeSWITCH
```

下列命令可以将 TTS 语音合成的结果保存到声音文件（hello.aiff）中，以便后续可以用其他播放软件播放：

```
$ say -o /tmp/hello.aiff Hello, Welcom to FreeSWITCH
```

注意，这里的声音文件我们使用了 AIFF 格式，因为该命令不支持 WAV 格式的文件。

我们可以使用 sox 软件包[⊖]中的 play 命令来播放该文件，如：

```
$ play /tmp/hello.aiff
```

也可以使用 sox 将其转换为 WAV 格式，如：

```
$ sox /tmp/hello.aiff /tmp/hello.wav
```

使用系统提供的 TTS 引擎的好处就是它一般比较容易支持更多的语种，例如，我们可以在 Mac 上增加中文语音的支持。Mac 系统默认不带中文的语音库，不过，可以自行安装。安装步骤是：打开“系统偏好”，依次选择“听写与语音”、“文本至语音”，在系统嗓音中选择自定义，就可以选择嗓音为“ Ting-Ting”的声音了，如图 11-1 所示，选择完毕后它会自动安装。



图 11-1　在 Mac 中安装中文 TTS 语音支持

在安装完中文支持并将默认的嗓音选为“ Ting-Ting”后，就可以使用下列命令让它说中文了：

```
$ say 欢迎使用 FreeSWITCH
```

当然，如果“ Ting-Ting”不是默认的嗓音，也可以在 say 命令后使用“ -v”指定使用的嗓音，如：

```
$ say -v Ting-Ting 欢迎使用 FreeSWITCH
```

在准备好中文的 TTS 引擎后，我们下一步就编译安装 mod_tts_commandline 模块，进入 FreeSWITCH 的源代码目录，执行如下命令：

⊖ sox（Sound eXchange）是一个跨平台的音频处理工具。该命令不是内置的，需要单独安装。参见：http://en.wikipedia.org/wiki/SoX。

```
# make mod_tts_commandline-install
```

安装完毕后，我们修改 tts_commandline 模块的配置文件，将 settings 部分修改为如下形式：

```
<settings>
    <param name="command" value="say -v ${voice} -o ${file}.aiff ${text}; sox ${file}.
aiff ${file}; rm -f ${file}.aiff"/>
</settings>
```

其中，FreeSWITCH 在调用该模块时会自动设置这几个变量：voice 为嗓音的名字；file 为声音文件的名字；text 为 TTS 文本；rate 为 Channel 的采样率。我们通过 FreeSWITCH 传过来的参数生成一个命令（command），该命令实际上由 3 个以分号分开的命令组成。第 1 个命令调用操作系统提供的 say，以便从 FreeSWITCH 传过来的文本（text）产生一个声音文件（$file）。需要指出，该模块默认的扩展名是 .wav，而 Mac 上的 Say 并不支持 .wav 格式的声音文件，所以，我们先生成一个 .aiff 文件，然后在第 2 条命令中使用 sox 将该 .aiff 文件转换为 .wav 格式的文件，并在第 3 条命令中使用 rm 将原先的 .aiff 删除（FreeSWITCH 在使用完毕后会自动删除 .wav 文件）。

然后到 FreeSWITCH 中加载该模块：

```
freeswitch> load mod_tts_commandline
```

接着，就可以用如下的 Dialplan 让 FreeSWITCH 说中文了（注意，为了支持中文，Dialplan 的配置文件应该使用 UTF-8 编码）：

```
<extension name="TTS">
    <condition field="destination_number" expression="^1234$">
        <action application="answer"/>
        <action application="set" data="tts_engine=tts_commandline"/>
        <action application="set" data="tts_voice=Ting-Ting"/>
        <action application="speak" data="欢迎使用 FreeSWITCH"/>
    </condition>
</extension>
```

2. 使用 Windows 平台上的 TTS 功能

在 Windows 平台上，也可以使用 Microsoft 提供的 TTS API 来使用 TTS 功能。

首先，建立以下 VBS 脚本，存放到一个目录下，如 C:\src\say.vbs。该脚本的内容如下：

```
set spvoice = CreateObject("SAPI.SpVoice")
set spfilestream = CreateObject("SAPI.SpFileStream")
set args = Wscript.Arguments
wavfile = Replace(args(0), "/", "\")
text = args(1)
Wscript.echo wavfile
spfilestream.open wavfile, 3
set spvoice.AudioOutputStream = spfilestream
spvoice.Speak text
spfilestream.close
```

该脚本首先使用 CreateObject 创建一个 SAPI 的语音对象 spvoice，然后创建一个文件对象 spfilestream，该文件对象将打开一个声音文件准备写入，接下来把 spvoice 的音频输出

（AudioOutputStream）指定为该文件对象，最后调用 spvoice 的 Speak 方法进行 TTS 转换，并把结果写入声音文件中。

我们可以先在命令行中进行实验，如：

```
C:\src> cscript say.vbs test.wav 欢迎使用 FreeSWITCH
```

上述命令将产生声音文件 test.wav，我们可以双击该文件听一下内容是否正确。如果听不到中文，那么很可能是没有安装支持中文的语音包。可以选择“开始”菜单→“控制面板”→“语音”项核实一下。如果在弹出的窗口的“语音选择”栏中只有“Microsoft Sam”一项可供选择，那表示没有中文的语音支持，需要单独安装。笔者测试时使用的是 Neospeech 提供的语音库（见图 11-2），另外，据说 Microsoft Office 中也有中文语音库的支持。如何在 Windows 上安装中文的语音支持超出了本书的范围，读者可以自行查阅相关资料。



图 11-2 在 Windows 上选择一个中文语音

如果一切正常，接下来就是配置 conf\autoload_configs\tts_commandline.conf.xml 文件了，将其中的 command 参数配置为如下形式：

```
<param name="command" value="cscript c:\src\say.vbs ${file} ${text}"/>
```

在 Windows 上，该模块默认也是不编译的，甚至没有一个 VC 的工程文件。如果要编译该模块，我们需要自己生成一个工程文件。在 Windows 上生成一个工程文件的步骤比在 UNIX 类平台上要复杂一些。但对于熟悉 VC 的人来说，生成一个工程文件应该不是很难。在此，我们还是找一个简单的方法。步骤如下：

1）找一个差不多的工程文件复制过来，然后稍做修改。笔者使用的是 VS2010 Express 版，因而将 application\mod_fsv 目录中的 mod_fsv.2010.vcxproj 工程文件复制到 asr_tts\mod_tts_commandline 目录中，并改名为 mod_tts_commandline.2010.vcxproj。接着，用记事本或其他文本编辑器打开该文件，将所有的 mod_fsv 替换为 mod_tts_commandline 就完成了。

2）打开 FreeSWITCH 的 Solution 文件 Freeswitch.2010.express.sln，如图 11-3 所示，右击“解决方案资源管理器”，在弹出的快捷菜单中选择“解决方案 Freeswitch.2.10...”，依次选择“添加”→“现有项目”，在弹出的窗口中找到我们刚刚建立的 mod_tts_commandline.2010.vcxproj，然后单击“确定”按钮就可以将该工程文件加入解决方案了。接着在“解决方案资源管理器”中找到“mod_tts_commandline”一项，右击，然后在弹出的快捷菜单中选择“生成（U)”命令编译该模块。

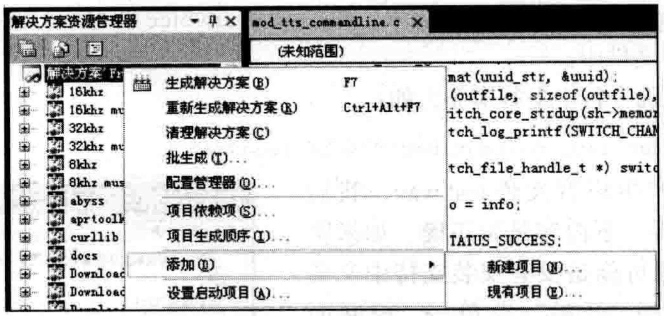


图 11-3 将 mod_tts_commandline 工程加入解决方案

编译完毕后就可以在 FreeSWITCH 中加载该模块了。剩余的步骤与在 Mac 上的操作是类似的，读者可以自己练习一下。

3. 使用 Linux 上的 TTS 功能

在 Linux 上，也有支持中文的 TTS 实现，不过笔者没有详细测试过。当然，如果读者运行的 FreeSWITCH 具有 Internet 环境，也可以使用 Google Translate 提供的 API 自己写一个脚本来与 mod_tts_commandline 集成。这部分内容留给读者自行练习，在此就不多介绍了。我们将在 11.7.4 节介绍 Google Translate。

11.7.3 MRCP

MRCP（Media Resource Control Protocol）㊀是一个支持访问网络上的媒体资源的协议。它的典型应用就是 TTS 和 ASR（自动语音识别）。MRCP 协议有两个版本，其中，v1 ㊁版使用 RTSP 协议进行媒体资源的协商，v2 ㊂版则使用 SIP 协议（这是我们看到的 SIP 协议的另外一个用处）。UniMRCP ㊃是 MRCP 协议的一个开源实现，它支持 v1 和 v2 两个版本。

mod_unimrcp ㊄是在 UniMRCP 基础上在 FreeSWITCH 中实现的一个模块，它同时支持 TTS 和 ASR。一般来说，主流的 TTS 和 ASR 产品都支持 MRCP 协议。这类产品专业性比较强，一般都是商业版的软件。在写作本书时，没有找到合适的商业软件作为例子，因而在此就不对具体模块的对接配置做过多介绍了。不过，我们将在 17.8.2 节讲一个使用 MRCP 进行语音识别的例子，感兴趣的读者也可以参考相关的配置（配置方法都是类似的）。

如果使用 UniMRCP，在 Dialplan 中 TTS 引擎的写法是“unimrcp:”加上配置文件中配置项的名字，如：

```
<action application="set" data="tts_engine=unimrcp:tts_server1"/>
```

其他的语法都与我们前面讲的类似，在此就不再赘述了。

㊀ 参见 http://en.wikipedia.org/wiki/Media_Resource_Control_Protocol。
㊁ 参见 http://tools.ietf.org/html/rfc4463。
㊂ 参见 http://tools.ietf.org/html/rfc6787。
㊃ 参见 http://www.unimrcp.org/。
㊄ 参见 http://wiki.freeswitch.org/wiki/Mod_unimrcp。

11.7.4 Google Translate

Google 提供了一些优秀的工具，Google Translate（即谷歌翻译）就是其中之一。笔者在研究过程中偶然发现 Google Translate 有一个很酷的 TTS 功能。

在浏览器中访问以下 URL[⊖]就可以播放 "Hello and welcome to FreeSWITCH" 了：

```
http://translate.google.com/translate_tts?q=Hello+and+welcome+to+FreeSWITCH&tl=en
```

若要播放中文也很简单，只需要将 en 改成 zh（代表中文），URL 如下：

```
http://translate.google.com/translate_tts?q=欢迎使用 FreeSWITCH&tl=zh
```

在上面的测试中，读的中文没问题，只是似乎把英文也按中文的习惯来读了，听起来很不习惯[⊜]。我们后面再解决这个问题。

为了在 FreeSWITCH 中使用该功能，我们需要一个命令行版本的 API。如下列命令可以使用 Google Translate 把文字变成 MP3 声音文件：

```
curl -A "Mozilla" -o google_tts.mp3
"http://translate.google.com/translate_tts?tl=zh&ie=UTF8&q=%e6%ac%a2%e8%bf%8e%
e4%bd%bf%e7%94%a8FreeSWITCH"
```

其中，我们使用了 curl 命令行工具，它是一个 HTTP 客户端。-A 指定的是客户端使用的 User Agent，Google 好像不喜欢 curl，因此这里我们使用 Mozilla 替代；-o 指定的是输出的文件，这里输出到一个 MP3 文件；最后是要访问的 URL，我们已经将中文用 url_encode 进行编码了（注意，与浏览器中的 URL 相比，本 URL 中多了个 ie=UTF8 参数，它指定输入文本的编码类型，这里我们使用 UTF-8）。

以下的 wget 命令与上面的命令是等价的，读者可以自己对比相关参数的含义。

```
wget -A "Mozilla" -o google_tts.mp3 "http://translate.google.com/translate_tts
?tl=zh&q=%e6%ac%a2%e8%bf%8e%e4%bd%bf%e7%94%a8FreeSWITCH&ie=UTF-8"
```

理论上，有了上述命令行以后，我们就可以编写一个脚本，用 11.7 节讲到的方法通过 mod_tts_commandline 来使用 Google Translate 的 TTS 功能了。当然，读者也可以自己试一下，我们在这里就不重复该方法了。下面，我们讲两种更简便的方法。

首先，加载 mod_shout，以让 FreeSWITCH 支持 MP3。然后，使用如下的 Dialplan 就可以了：

```
<extension name="Free_Google_Text_To_Speech">
    <condition field="destination_number" expression="^1234$">
        <action application="answer" data=""/>
        <action application="playback"
           data="shout://translate.google.com/translate_tts?&tl=zh&ie=UTF-8&q=
```

⊖ 其中的 URL 字符串都是经过 url_encode 编码的，不熟悉的读者可以参考一下 http://zh.wikipedia.org/wiki/百分号编码。另外，这里有一个站长工具可以帮助进行这种编码或反编码（解码），参见 http://tool.chinaz.com/Tools/URLEncode.aspx。

⊜ 或许这也是 Google 可爱的地方。如它把 "FreeSWITCH" 读成类似 "埃弗啊夷塔八六埃四意吃"。当然，自己试一下就知道了。网上也有好多调侃（或者说调戏）Google Translate 的段子。

```
欢迎使用 +FreeSWITCH"/>
          </condition>
    </extension>
```

呼叫 1234 就应该可以听到测试的声音了。前面，我们在 URL 中没有使用“http://”，而是使用了“ shout://”。shout 代表 SHOUTcast ⊖协议，它是专门用于流媒体广播的，Google Translate 和 FreeSWITCH 中的 mod_shout 就直接支持该协议，因此我们可以直接使用它，以获得最好的效果。

上面我们提到，Google Translate 用中文方式读的英语不好听，这里我们要解决一下。目前看来，最简单也是最有效的办法就是将它拆成中文和英文分别读，具体如下：

```
<action application="playback"
   data="shout://translate.google.com/translate_tts?&tl=zh&ie=UTF-8&q= 欢迎使用 "/>
<action application="playback"
   data="shout://translate.google.com/translate_tts?&tl=en&q=FreeSWITCH"/>
```

当然，实际上 shout 也是 FreeSWITCH 实现的一个文件接口（File Interface)。除此之外，也可以使用 mod_httapi 提供的 HTTP 文件接口来解决上述问题，如（我们省略了较长的 URL)：

```
<action application="playback" data="http://translate.google.com/translate_tts?...
```

或者试一下 mod_vlc 中提供的 HTTP 文件接口：

```
<action application="playback" data="vlc/http://translate.google.com/translate_
tts?...
```

总之，不管以何种方式调用这些 API，殊途同归，最后都使用这些 API 生成我们需要的声音文件（或数据)，并播放出来。

11.7.5 TTS 小结

FreeSWITCH 支持比较丰富的 TTS 功能，也能很容易地与大部分 TTS 产品进行集成，而且它使用起来也比录音文件灵活。那么，什么情况下该使用 TTS，什么情况下使用录音呢？

坦白地说，这其实不是个技术问题，也没有标准答案，不过在两者之间进行选择时，我们可以考虑以下几个问题：

- 如果在开发或测试时，使用 TTS 会比较简单，因为文本内容也比录音文件直观。
- 即使再好的 TTS 产品，它的发音也像机器的发音，永远不如真人录音好听。
- 如果录音的人不懂得录音的技巧，录出来反而可能不专业，不好听。
- 如果语音提示需要经常改动，那最好用 TTS，因为 TTS 改起来会比较方便。更重要的是，它能保持统一的嗓音；而使用真人录音，可能找不到原先录音的那个人了。
- TTS 产品还是比较有技术含量的（如分词断句、多音字等)，所以一般来说好的 TTS 产品都比较贵。

当然，具体问题具体分析。熟练掌握了这些知识，相信大家在真正使用时都能做出正确

⊖ SHOUTcast 是一个流媒体的广播协议，参见 http://en.wikipedia.org/wiki/SHOUTcast。

的选择。就笔者的经验来看，在系统上配置好 TTS 后，以后写 IVR 之类的应用程序就比较简单了。因为直接使用 TTS，可以避免先录音，也不用确定录音文件名与实际内容的对应关系等，而且在 IVR 脚本或程序中直接嵌入文件内容，代码看起来也比文件名要直观得多，这样可以减少错误的发生。等到全部 IVR 应用都写好了以后，可以再选择是否使用商业的 TTS 软件（也许能增强朗读效果），或者找专业的人员录音实现。在本书后面的例子中，我们在需要声音文件的地方大部分都将使用 TTS 为例来进行讲解。

11.8　在呼叫失败的情况下向主叫用户播放语音提示

在传统的电话网络中，在呼叫失败的情况下交换机会给主叫用户一个有意义的语音提示，如“您拨打的电话正在通话中，请稍后再拨……”或“您拨打的电话暂时无法接通……”之类。在 FreesSWITCH 中也可以做到。

11.8.1　实现方法

若已经熟练掌握了前面的知识，则再来看具体的实现就觉得很简单了。在此我们在默认的配置上修改，以实现具体的功能。

在默认的配置中，对本地用户的路由都是在 conf/dialplan/default.xml 中的 Local_Extension 部分配置的。在呼叫失败时，会将电话转到 voicemail（语音信箱），提示用户可留言。为了能给主叫用户一个语音提示，我们可以在电话进入语音信箱之前做点文章，让它播放一个语音提示（当然播放完毕后也可以不进入语音信箱而直接挂机）。

找到 <extension name="Local_Extension"> 部分的最后几行：

```
<action application="bridge" data="user/${dialed_extension}@${domain_name}"/>
<action application="answer"/>
<action application="sleep" data="1000"/>
<action application="bridge" data="loopback/app=voicemail:default ${domain_
name} ${dialed_extension}"/>
```

其中，第一个 bridge 用于呼叫被叫号码，如果呼叫失败，则 Dialplan 会继续往下执行，执行顺序是：

- 应答（answer）；
- “睡”一会（sleep，即暂停 1 秒钟，其中 1000 的单位是毫秒）；
- 进入语音信箱（voicemail）。

到这里，我们只需要在最后一个 bridge 之前加入下面一行：

```
<action application="playback" data="/tmp/${originate_disposition}.wav"/>
```

重新打电话试一下，如果被叫忙（或拒接），则 originate_disposition 变量的值就会是 USER_BUSY（有的终端会返回 CALL_REJECTED），而如果用户没注册就是 USER_NOT_REGISTERED 之类的。通过增加上面一行，笔者在拨打一个不存在的用户时，看到如下的日志：

```
EXECUTE sofia/internal/1002@192.168.7.5 playback(USER_NOT_REGISTERED.wav)
[ERR] mod_sndfile.c:202 Error Opening File [/tmp/USER_NOT_REGISTERED.wav][System
error : No such file or directory.]
```

上面的日志提示找不到指定的文件。当然这个问题很好解决，我们录一个放上去就可以了，内容可以是“您拨打的电话暂时无法接通……”。当然，对于其他情况，如用户忙，我们也可以录一段提示并放到 /tmp/USER_BUSY.wav 文件中。

11.8.2 进阶

不过，呼叫失败的原因可能有很多，我们不可能录上所有的声音文件，此时有两种方法可以解决这个问题。

1）使用一个 Lua（或其他语言）脚本。

我们可以不增加刚才 playback 一行的配置（即上一小节提到的在最后一个 bridge 之前加入的代码），而是增加如下的行：

```
<action appliction="lua" data="/tmp/xxx.lua"/>
```

在 Lua 脚本中可以取到 originate_disposition 变量，从而可以使用 if...then...else 之类的逻辑选择播放各种声音文件（即只针对有限几种常用的情况播放特定的声音，其他的一律播放同一个文件）。关于 Lua 开发的知识我们将在第 16 章讲到，在这里就不多讲了。

2）当然，如果不想使用 Lua，也可以使用 Dialplan 配置做到。实际上，FreeSWITCH 的 Dialplan 功能是非常强大的，你只需要使用如下配置将呼叫转到播放不同声音文件的 Dialplan Extension：

```
<action application="transfer" data="play-cause-${originate_disposition}"/>
```

然后创建如下 Dialplan Extension，以匹配上面转移的目的地：

```
<extension name="Local_Extension_play-cause">
    <condition field="destination_number" expression="^play-cause-USER_BUSY$">
        <action application="playback" data="/tmp/user-busy.wav"/>
    </condition>
</extension>

<extension name="Local_Extension_play-cause">
    <condition field="destination_number" expression="^play-cause-USER_NOT_
REGISTERED$">
        <action application="playback" data="/tmp/user-not-registered.wav"/>
    </condition>
</extension>

<extension name="Local_Extension_play-cause">
    <condition field="destination_number" expression="^play-cause(.*)$">
        <action application="log" data="WARNING hangup cause: $1"/>
        <!-- 对于其他各种情况，播放该文件 -->
        <action application="playback" data="/tmp/sounds/unknown-error.wav"/>
    </condition>
</extension>
```

这里的 Dialplan 配置即根据转移的目的地参数的不同播放不同的声音文件。读者可根据前面的知识自行分析。

11.8.3 使用 TTS

前面我们已经学会了强大的 TTS 功能，也讲到了 TTS 的各种好处，所以这里我们用 TTS 做语音提示。在此，我们使用前面讲过的 Phrase Macro 功能来实现各种提示音。

首先，我们在 11.7.2 节的基础上[⊖]，在 conf/lang/zh/demo 目录下创建以下脚本，可以将文件命名为 book-demo.xml，内容如下：

```
<include>
  <macro name="USER_BUSY">
    <input pattern="(.*)">
      <match>
        <action function="speak-text" data="分机 $1 正在通话，请稍后再拨"/>
      </match>
    </input>
  </macro>
  <macro name="USER_NOT_REGISTERED">
    <input pattern="(.*)">
      <match>
        <action function="speak-text" data="分机"/>
        <action function="say" data="$1" method="iterated" type="number"/>
        <action function="speak-text" data="暂时无法接通，请稍后再拨"/>
      </match>
    </input>
  </macro>
</include>
```

可以看出，我们上面配置了两个宏（Macro），第一个用于在用户忙的时候播放；第二个在用户未注册的时候播放。其中，name 为 Macro 的名字，pattern 会使用正则表达式匹配输入的参数（类似于 Dialplan 中的正则表达式）。在这里，我们会将被叫号码作为输入参数。也就是说，如果呼叫分机 1009 失败，就会播放“分机 1009 正在通话，请稍后再拨”。当然，我们在第二个 Macro 中使用了 Say 来播放被叫号码，只是为了演示另外一种实现方式。

配置好上述文件后，在 FreeSWITCH 控制台上执行 reloadxml 使之生效。然后，在 Dialplan 中，就可以将 11.8.1 节的 playback 的参数换写成以下这样：

```
<action application="playback" data="phrase:${originate_disposition}:$1"/>
```

其中，以“phrase:”开头的文件参数表示这里要播放一个 Phrase Macro，冒号后面跟的是参数，即我们在 Dialplan 正则表达式匹配中获得的被叫号码。如果被叫是 1009，并且在被叫忙的情况下，上述配置展开后最终的结果是这样的：

```
<action application="playback" data="phrase:USER_BUSY:1009"/>
```

当然，也可以在 Dialplan 中直接使用 phrase App，下面的代码与上述配置是等价的（不

⊖ 注意在 zh.xml 中需要配置正确的 TTS 引擎，如笔者配置的是“tts-engine="tts_commandline" tts-voice="Ting-Ting"”。

同的是，这里参数的分隔符是逗号）：

```
<action application="phrase" data="${originate_disposition},$1"/>
```

执行 reloadxml 后，再次拨打一个失败的电话试一下，就可以听到用 TTS 功能播放的提示音了。

当然，有了 Phrase Macro，我们的配置方式就更灵活了，如我们可以只用一个 Phrase Macro 来搞定所有的情况。先配置如下 Dialplan：

```
<action application="phrase" data="HangupCause,${originate_disposition}"/>
```

为了配合上述 Dialplan，我们需要在 Phrase Macro 中配置一个名为 HangupCause 的 Macro，并将 originate_disposition 作为参数传入。然后我们再来配置该 Macro，同一个 Macro 中配置多个 input 可以针对不同的参数输入执行不同的动作（action），具体如下：

```
<macro name="HangupCause">
    <input pattern="USER_BUSY">
        <match>
            <action function="speak-text" data="您拨的电话正在通话，请稍后再拨"/>
        </match>
    </input>
    <input pattern="USER_NOT_REGISTERED">
        <match>
            <action function="speak-text" data="您拨的电话不在线，请稍后再拨"/>
        </match>
    </input>
    <input pattern="(.*)">
        <match>
            <action function="speak-text" data="您拨的电话暂时无法接通，请稍后再拨"/>
        </match>
    </input>
</macro>
```

可以看出，该 Phrase Macro 对前两种情况播放不同的语音提示，对以后所有的情况都播放同一个语音提示，比在 11.8.2 节中用 Dialplan 方式实现显然更简洁，也更容易理解。

当然，最后这一个例子在播放提示时没有播放被叫号码。如果读者也想在提示音中包含被叫号码，可以与 originate_disposition 参数一起传到 Phrase Macro 中，使用一个分隔符隔开（如“!”），这样就可以再用类似“^(.*)!(.*)$”这样的正则表达式将两个参数取出来了。具体的实现留给读者练习，在此就不多讲了。

11.8.4 原理

当然，在被叫失败后播放我们上面指定的提示音或 TTS 是有一个前提的——在 Dialplan 的第一个 bridge 之前要有以下两行：

```
<action application="set" data="hangup_after_bridge=true"/>
<action application="set" data="continue_on_fail=true"/>
```

其中，第一行的作用是：在一个成功的 bridge 之后，如果被叫用户挂断电话，我们也没有必要再向主叫用户播放提示音了，因此可以直接挂机（Hangup After Bridge 的字面意

思就是“在 Bridge 成功后挂断电话”)。当然，这一行也可以没有，这就需要在后面检查 originate_disposition 变量值的时候，发现如果是“NORMAL_CLEARING”(表示呼叫正常释放)，再决定是否播放相关语音文件或挂机。

第二行的作用是：如果呼叫失败(fail，如空号，拒接等)，让 Dialplan 继续往下执行；否则(值为 false 的情况)Channel 在第一次呼叫(bridge)失败时就挂机了。该变量的值除了是笼统的“true”以外，还可是各种用逗号分隔开的挂断原因值。例如，下面的配置表示只有遇到用户忙或无应答这两种情况才播放语音，其他的就直接挂机。

```
<action application="set" data="continue_on_fail=USER_BUSY,NO_ANSWER"/>
```

常见的挂断原因取值除了上述代码中的外还有 NORMAL_TEMPORARY_FAILURE(普通临时失败)、TIMEOUT(超时)、NO_ROUTE_DESTINATION(空号 / 无法路由)等几种。更详细的失败原因可以参见 http://wiki.freeswitch.org/wiki/Hangup_causes。

总之，通过上述两个参数就可以在呼叫失败的时候让 Dialplan 正常往下进行，进而有机会检查上一次呼叫失败的原因，并播放相关的语音提示。

11.9　实现呼叫前转业务

在传统的电话系统中，有一项基础的增值业务称为呼叫前转。它的使用方法是：通过拨打一个特定的功能码(一般是“*57*”)登记欲转移到的电话号码，以后所有呼叫都会转移到该号码上。具体来讲，如分机 1002 是 Alice 办公室的电话，由于她临时需要到另一个办公室工作，而另一个办公室有一部电话 1007，因此，Alice 在 1002 上拨打 *57*1007# 就可以登记呼叫前转业务，这样如果以后有人呼叫 1002，就会自动转移到 1007 上，这样 Alice 就不会错过任何电话了。当她从另一个办公室回来时，她可再在 1002 上拨打 #57# 取消该业务，以后再有来话 1002 就会直接振铃了，而不会再发生转移了。

在传统的交换机上，需要为 1002 分机开通一个权限的，然后该分机才可以拨打 *57* 或 #57# 进行登记和取消。当然，我们在这里先不考虑这些权限问题，认为所有用户都有该权限就可以了。

为了实现业务登记，我们先建立以下路由：

```
<extension name="call_forward_unconditional_set">
  <condition field="destination_number" expression="^\*57\*([^#]*)#?$">
    <action application="hash" data="insert/${domain_name}-cfwdu/${username}/$1"/>
    <action application="answer"/>
    <action application="playback" data="您登记的业务已经成功.wav"/>
  </condition>
</extension>
```

可以看出，正则表达式“^*57*([^#]*)#?”会匹配“*57*1007#”，并将电话号码存储在变量 $1 中。这里，考虑到有些话机无法输入末尾的 # 号键(有些话机会认为按 # 表示号码输入完毕，因而送给 FreeSWITCH 的被叫号码不包含 # 号)，我们通过正则表达式中的

“?”把它标志为可选的。也就是说，如果用户拨打“*57*1007”也会匹配到这里。

在电话到达后，我们使用 hash[⊖] App 将该信息存储在系统内存中的一个哈希表中。hash 的参数是以“/”隔开的几项：insert 表示向哈希表中插入数据；${domain_name}-cfwdu 表示一个域（realm），该域是我们自己定义的，其中的 cfwdu 是无条件呼叫前转（Call Forward Unconditional）的英文简写形式，用于标志我们这里的呼叫前转业务；${username} 是一个键（key），在本例子中，它就是主叫用户 1002；$1 是该键的值（value），在此它等于 1007。

当然，登记成功后，我们不会忘记使用 playback 播放一个友好的语音提示。

通过下面 Dialplan Extension，可以在 1002 拨打 #57# 时将登记的信息删除：

```
<extension name="call_forward_unconditional_unset">
  <condition field="destination_number" expression="^#57#?$">
    <action application="hash" data="delete/${domain_name}-cfwdu/${username}"/>
    <action application="answer"/>
    <action application="playback" data="您登记的业务已经取消.wav"/>
  </condition>
</extension>
```

同样，我们也不要求被叫号码最后必须有 # 号，就算没有上面的正则表达式也会匹配 #57。可以看出，hash App 的参数由 insert 换成了 delete，表示我们要删除一个键。

在实现了登记的删除后，我们就在正常的路由中进行检查了。来看下面的 Dialplan 设置：

```
<extension name="call_forward_unconditional_check" continue="true">
  <condition>
    <action application="info" data=""/>
    <action application="set" inline="true"
        data="cfwdu=${hash(select/${domain_name}-cfwdu/${destination_number})}"/>
  </condition>
</extension>
```

上述 Dialplan 定义了一个绝对的条件（因为 <condition> 标签没有其他属性），因而它总是会被匹配到。在电话路由到这里后，我们先使用 info App 在日志中打印一些调试信息，以方便查找问题。然后，通过 set App 设置一个 cfwdu 变量，该变量的值是从哈希表中使用 select 取出的。如果用 *57* 进行了登记，则表示它就有值，否则就是一个空字符串。在此，如果有人拨打 1002，则这里哈希表的键就是以 ${destination_number} 标志的，即 1002。

注意，此处的 inline="true" 参数非常关键。有了它，这一行的 set 命令在 Dialplan 解析的阶段就会执行，这样到后面我们才能根据该变量来进行后续的条件判断。

在此，我们也可以用下面的 hash API 命令进行检查（192.168.7.5 是笔者电脑上 ${domain_name} 的值）：

```
freeswitch> hash select/192.168.7.5-cfwdu/1002

1007
```

可以看出，我们确定可以在域 192.168.7.5-cfwdu 中通过 1002 这个键找到 1007 这个值（当然，前提是 1002 已经拨打过“*57*1007#”进行登记）。

⊖ 用于在 FreeSWITCH 中使用内存中的哈稀表存储数据，参见 http://wiki.freeswitch.org/wiki/Mod_hash。

然后，继续向下路由，直到下面的 Dialplan Extension：

```
<extension name="call_forward_unconditional_check2" continue="true">
  <condition field="${cfwdu}" expression="^10[01][0-9]">
    <action application="bridge" data="user/${cfwdu}"/>
  </condition>
</extension>
```

该 Extension 检查的条件是 cfwdu 这个变量的值（就是我们上一步中 set 的），如果它的值匹配一个分机号（登记后它将匹配 1007），我们就执行 bridge，它的参数在本例中将是 user/1007，因而电话就"转移"到分机 1007 上去了。如果不匹配，Dialplan 就会继续向下执行，跟没有登记是一样的。

在本例中，我们用了内存中的哈希表来存储登记的号码，如果 FreeSWITCH 重启，这些值都会消失。如果希望在重启后也能保存这些值，可以使用持久化的数据库存储。mod_db 实现了持久化的数据库存储，与 hash 类似，它提供了" db insert"、" db delete"、" db select" 等。详情请参见 http://wiki.freeswitch.org/wiki/Mod_db。

此外，我们在本例中仅实现了无条件呼叫前转，对于其他转移方式，如遇忙转移和无应答转移等，可以根据本节的内容配合我们在上一节中讲到的方法进行实现。具体的实现方法留给读者自行练习，在此不再赘述。

11.10　小结

本章我们首先通过一些实例讲解了 FreeSWITCH 中最基本的用户、IVR 及时间相关的 Dialplan 的配置，涵盖了最基本的配置、Application 和路由相关的知识。其次，我们通过录音和放音的例子讲解了 FreeSWITCH 的文件接口，帮助大家了解了 FreeSWITCH 所支持的各种文件格式。再次，对于我们中国人来讲，在 FreeSWITCH 中能否使用中文语音提示肯定是大家最关心的问题，对此我们通过一些例子说明了各种实现方法。然后，我们介绍了 TTS 技术以及如何使用中文的 TTS。通过使用 TTS，可以快速设置语音提示和 IVR 等（至少省了录音的环节），而且我们在这里讲起来也更直观一些。最后，我们通过讲解呼叫失败后的处理以及呼叫前转的实现两个例子进一步讲解了 Dialplan 的配置、各种 App 的灵活运用等，并顺便讲了 FreeSWITCH 中用于临时及持久化存储的 hash 以及 db 等相关内容。

通过这些实际的例子，希望读者能加深对各种配置文件的认识，并能熟练掌握这里介绍的配置步骤，举一反三，进而对其他类似的模块和配置也能很快入手。

第 12 章

高级功能与配置实例

在上一章，我们学习了 FreeSWITCH 的一些基本功能与配置实例。本章我们来学习一些比较高级的功能，如呼叫中心应用、多人会议、与数据库的连接、话单及计费等。

12.1 使用 mod_fifo 实现简单呼叫队列

在呼叫中心应用中，有一个很常用的功能——ACD（Automatic Call Distributioin，自动电话分配）。一般来说，我们这里所说的呼叫中心都是客服型呼叫中心，即有一组话务员（称为座席，英文称为 Agent）在等待为用户服务。当有电话呼入时，先将用户放入一个队列进行排队（因为可能同时有大量用户呼入），然后按一定的策略选择一个空闲的座席为用户服务（将来话从队列中取出并与该座席通话）。这种通过一定策略为来话分配空闲座席的功能就称为 ACD。

mod_fifo 模块实现了一些简单的 ACD 功能，下面我们来看一下。

12.1.1 呼叫停泊与取回

一般来说，普通的电话呼叫会使用 bridge App 来桥接两条腿。但在电话分配中，一般采用停泊与取回的方式进行电话搭接。顾名思义，停泊跟泊车类似，即在有来话时，先将来话停靠在一个泊位上（一般还会放点音乐），然后通过某种算法找到一个空闲的座席，该座席拨打一个与该泊位相关的号码，就可以将该泊位上的电话取回，双方进行通话。

FreeSWITCH 默认的 Dialplan 中提供了使用这种方式进行电话分配的例子。我们先来看停泊的部分，Dialplan 设置如下：

```
<extension name="park">
    <condition field="destination_number" expression="^5900$">
        <action application="set" data="fifo_music=$${hold_music}"/>
        <action application="fifo" data="5900@${domain_name} in"/>
    </condition>
</extension>
```

拿起电话拨打 5900，就可以将电话停在一个泊位上。从上面的 Dialplan 中可以看出：首先，它使用 set 设置了一个 fifo_music 的通道变量，该变量指定电话停靠在泊位上时播放的音乐。然后，使用 fifo App 将电话放入一个先入先出的队列中（fifo 的意思就是 First In, First Out，即先入先出），该队列的名字是 5900@${domain_name}，一般来说 ${domain_name} 变量的值是 FreeSWITCH 所在服务器的 IP 地址。参数 in 表示是入队。

当有电话呼入时，可以使用 fifo list 命令显示当前队列的状态，具体如下：

```
freeswitch> fifo list

<fifo_report>
    <fifo name="5900@192.168.7.9" consumer_count="0"
    caller_count="1" waiting_count="1" importance="0"
    outbound_per_cycle="0" ring_timeout="0" default_lag="0"
    outbound_priority="5" outbound_strategy="ringall">
        <outbound></outbound>
        <callers>
            <caller uuid="a8cbbbd0-91a8-4f3f-be6e-1741e25deb1d"
                    status="WAITING" caller_id_name="1002"
                    caller_id_number="1002"
                    timestamp="2013-10-14 21:10:18"
                    position="1" slot="0">
            </caller>
        </callers>
    <consumers></consumers>
    <bridges></bridges>
    </fifo>
    ...
```

该列表是以 XML 格式显示的。其中，我们可以看到一个 fifo 标签，它的名字是 5900@192.168.7.9，它里面有一个 caller（即主叫），并显示了该 caller 的当前的状态是 WAITING（即在队列中等待服务）。

fifo 是一个“生产者——消费者”模型，即来话（Caller）相当于生产者（Producer），而座席（Agent）则称为消费者（Consumer），它对来话进行服务即相当于“消费”生产者生产的内容。

我们接着来看下列的 Dialplan，它也是来自默认的配置：

```
<extension name="unpark">
    <condition field="Destination_number" expression="^5901$">
        <action application="answer"/>
        <action application="fifo" data="5900@${domain_name} out nowait"/>
    </condition>
</extension>
```

通过上述 Dialplan，如果座席端通过某种方式（如业务界面上的通知，可以使用 HTML 等其他消息通知机制实现）知道队列里有一个来电需要服务，座席话务员就可以（手工或自动）拨打 5901 将该来电从队列里出取来，然后双方进行通话。可以看出，这里 Dialplan 的配置与前面入队时是类似的，所不同的是使用了 out 参数代表出队，另外有一个附加的 nowait 参数，它表示如果队列中没有电话在等待（可能这时候主叫已经等不及而挂机了），

即没有人需要服务时，座席端的电话就没必要在这里等待了。当然，如果把该参数改成 wait，或者不加该参数（默认值为 wait），那么座席端会在这里继续等待，直到队列里来了一路通话，它便可以立即得到服务。

12.1.2 配置座席

本节我们来介绍座席的具体配置方法。

1. 静态座席的配置

上一节我们讲了 fifo 入队及出队的简单原理。一般情况下，fifo 是不需要配置的，如果在呼叫中需要一个 fifo，FreeSWITCH 就会自动创建（如上面的 fifo 名字可以改成任意的其他字符串）。但有时候，我们也可能希望配置一些静态的座席用于呼叫分配。

mod_fifo 的配置文件是 conf/autoload_configs/fifo.conf.xml。它默认的配置中有一个 cool_fifo 的例子，为了简单起见，我们自己配置一个 fifo 队列，命名为 book，具体如下：

```
<fifos>
    <fifo name="book" importance="0">
        <member timeout="60" simo="1" lag="5">user/1005</member>
        <member timeout="60" simo="1" lag="5">user/1006</member>
    </fifo>
</fifos>
```

其中，我们为 book 队列配置了两个 member（1005 和 1006），每个 member 相当于一个座席。user/1005 为该 member 的呼叫字符串，即队列中有来话时怎么呼叫该座席；timeout 表示呼叫超时值；simo 表示最大能服务的呼叫的数量；lag 表示该座席在接听一个呼叫后，隔多长时间（秒）才可以再接收下一个呼叫。

使用如下命令可以使刚才的配置生效：

```
freeswitch> reloadxml
freeswitch> fifo reparse
```

然后，可以使用 fifo list book 查看一下配置的结果。该命令显示的结果比我们配置的会多出一些参数，这里我们就不详细介绍了。

我们可以使用如下 Dialplan 将来话路由到我们刚刚配置的 fifo。

```
<extension name="fifo-test">
    <condition field="destination_number" expression="^(1234)$">
        <action application="set" data="hold_music=$${hold_music}"/>
        <action application="fifo" data="book in"/>
    </condition>
</extension>
```

当有用户拨打 1234 时，便进入我们刚才配置的 fifo，听等待音乐。然后，1005 和 1006 就会轮流振铃（笔者在试验中，拒接其中一个，另一个就振铃）。

当将队列中的来话分配到座席时，fifo 使用轮循的方式分配，因而它是“公平”的，即不会出现某些座席“忙死”而其他座席“饿死”[⊖]的情况。

⊖ 闲着没事干的情况称为“饥饿”。

2. 动态座席的配置

在实际的呼叫中心应用中，座席的配置往往是动态的，即一个话务员上班时，会执行一个“签入”的动作，说明他（她）上班了，准备好接电话了；而在中途休息时，或者下班时执行“签出”，这样队列里再有电话进来就不会分配到该座席了。

mod_fifo 提供了一个 fifo_member 命令可以动态增加和删除座席，如我们可以用如下命令再增加两个座席：

```
freeswitch> fifo_member add user/1007
freeswitch> fifo_member add user/1008
```

如果删除座席，则可以使用如下命令（注意，最后一个参数呼叫字符串必须与增加的时候严格相同）：

```
freeswitch> fifo_member del user/1008
```

熟悉了这些命令后，就可以使用 Lua 脚本或 Event Socket 等与其他系统集成了，如可以在业务系统的网页上集成一些按钮，按这些按钮就可以进行签入或签出了。

当然，在一些简单的呼叫中心中可能没有业务系统，甚至没有条件做网页。在这种情况下，也可以用我们学过的知识，使用话机签入签出。如下面的 Dialplan 就可以做到：

```
    <extension name="Agent Login">
        <condition field="destination_number" expression="^5902$">
            <action application="answer"/>
            <action application="set" data="result=${fifo_member(add book user/
${user_name})}"/>
            <action application="playback" data="ivr/ivr-you_are_now_logged_in.wav"/>
        </condition>
    </extension>

    <extension name="Agent Logout">
        <condition field="destination_number" expression="^5903$">
            <action application="answer"/>
            <action application="set" data="result=${fifo_member(del book user/
${user_name})}"/>
            <action application="playback" data="ivr/ivr-you_are_now_logged_out.wav"/>
        </condition>
    </extension>
```

通过使用上述 Dialplan，就可以在上班时在自己话机上拨打 5902 进行签入，下班时拨打进行 5903 签出，它们都是使用了我们上面讲过的 fifo_member API 命令实现的。

3. offhook 座席

以上配置的座席中，不管是静态配置的还是动态配置的，都称为 onhook（挂机）座席。这种座席的特点是当队列中来了电话后再去呼叫座席。而与之相对的另一种座席称为 offhook（摘机）座席，这种座席会事先呼入队列并等待（一般听等待音乐），当有来电时就可以立即接听，省去了呼叫座席的时间，因而能更迅速为客户提供服务。这在比较繁忙的呼叫中心中也有助于提高效率。

其实我们把 12.1.1 节从队列中取回电话的例子稍作修改，就可以支持 offhook 座席。比

如，我们建立下面的Dialplan，当座席呼叫5902时，就会执行fifo App，它会尝试从队列中取出一个呼叫。而此时如果队列中没有呼叫在等待服务，则由于此处我们使用了wait参数（对比一下12.1.1节中使用的是wait），因此该座席会一直停在这里等待，直到队列中有新的电话进入。该Dialplan的设置如下：

```
<extension name="unpark">
    <condition field="destination_number" expression="^5902$">
        <action application="answer"/>
        <action application="fifo" data="5900@${domain_name} out wait"/>
    </condition>
</extension>
```

当服务完成后，该座席就又变成等待状态，继续等待下一个呼叫的到来。

12.1.3 fifo

本节我们来详细介绍一下fifo。

1. 相关的通信变量

在mod_fifo中，有许多相关的通道变量可以改变它的行为。通过有效地使用这些通道变量，往往能配置出比较实用的功能。

如每个fifo都有10个优先级队列，在进行入fifo前，可以使用fifo_priority变量指定来话的优先级，高优先级的来话将排在队列的前面（默认的优先级是5）：

```
<action application="set" data="fifo_priority=1" />
<action application="fifo" data="book in" />
```

在队列中有多个电话排队时，座席这一端也可以随意取出一个电话进行接听。当然，如果达到这种效果需要通过其他手段获取到队列中所有成员的UUID，如下面的设置中，我们在从队列中取出（out）电话前先把fifo_bridge_uuid变量设置为想取出的Channel的UUID，然后就可以取出与特定的UUID相关联的通话了：

```
<action application="set" data="fifo_bridge_uuid=9c3a1fee-fae2-4355-bceb-5064f2107983" />
<action application="fifo" data="book out" />
```

在下面的例子中，我们使用了很多的通道变量。该例子基于多年前笔者实现的一个呼叫中心应用。我们先看看Dialplan中的配置（为了方便讲解，我们增加了行号）：

```
01 <action application="answer"/>
02 <action application="set" data="fifo_music=$${hold_music}"/>
03 <action application="set" data="tts_engine=tts_commandline"/>
04 <action application="set" data="tts_voice=Ting-Ting"/>
05 <action application="set" data="fifo_chime_list=say:座席全忙，继续等待请按1，转其他服务请按2"/>
06 <action application="set" data="fifo_chime_freq=15"/>
07 <action application="set" data="fifo_orbit_exten=1007:45"/>
08 <action application="set" data="fifo_orbit_dialplan=XML"/>
09 <action application="set" data="fifo_orbit_context=default"/>
10 <action application="set" data="fifo_orbit_announce=say:为保证服务质量，您的电话可能被录音"/>
```

```
11 <action application="set" data="fifo_caller_exit_key=2"/>
12 <action application="set" data="fifo_caller_exit_to_orbit=true"/>
13 <action application="set"
    data="fifo_override_announce=say:tts_commandline:Ting-Ting:来电号码${caller_
id_number}"/>
14 <action application="fifo" data="book in"/>
```

当有来电进来时，我们会将电话路由到这里的 Dialplan 处理。其中，第 1 行先对来话进行应答；第 2 行设置来话在队列中听到的保持音乐；第 3 行和第 4 行设置 TTS 引擎的参数（因为使用 TTS 比使用声音文件讲起来更方便一些）；第 5 行设置一个声音，它会每隔 15 秒（在第 6 行设置）播放一次，以给用户比较好的体验[⊖]；第 7 行，设置一个分机号，如果等待超时（冒号后面指定超时时间，即 45 秒），会转到该号码；第 8 行和第 9 行分别是与第 7 行配合的 Dialplan 和 Context；第 10 行，设置一个声音，如果座席接听，则先给来话用户播放一段提示音；第 11 行，设置一个按键，配置第 5 行；第 12 行，实现在用户按了第 11 行的按键的情况下，转到第 7 行的号码上（否则的话将会退出 fifo）；第 13 行，是在座席接听电话后，在与来电客户通话前播放给座席的，这里我们播放了来电的号码[⊜]；第 14 行，这里是最重要的，即将来话送入 book 这个 fifo 队列中（如果没有这一行的话上面所有的设置就都白费了）。

还有一些其他变量是在 fifo 是执行过程中自动产生的，如 fifo_position 表示来话在当前队列中的位置；fifo_serviced_by 或 fifo_serviced_by 表示来话被哪个 Channel（通过 UUID 可以进而找到相关的座席）服务了。

当然，该模块还有其他相关的变量，限于篇幅，我们就不一一讲解了。有需要的读者可以参阅 http://wiki.freeswitch.org/wiki/Mod_fifo。

2. 相关事件

当 fifo 相关的状态发生变化，会产生一些 Subclass 为 fifo::info 的 CUSTOM 事件。通过这些事件，可以获取 fifo 中的详细信息，如哪个座席为哪个来话进行了服务等。

事件消息中，有一个 FIFO-Action 消息头标志了事件实际的动作。在笔者的试验中，一通来话从入队到挂机，大致经过以下几个事件：

- push：来话入队时产生。
- pre-dial：呼叫座席之前产生。
- post-dial：呼叫座席之后产生。
- consumer_start：消费者（即座席）开始。

⊖ 注意，在本例中，我们仅设置了按键“2”对应的动作，没有设置按键“1”，但提示音中包含了“1”。实际上，“1”本自是不存在的，即按与不按结果都一样。但如果这里提示“转其他服务请按 2，继续等待请不按”的话，显得有点别扭。当然，也可以把这里的提示音改成“请继续等待，或按 2 转其他服务”。总之，这是一个用户体验问题，而不是技术问题。我们在此主要是以这些例子来讲解各变量的含义，实际的提示音可以在实际应用场景中根据需要进行调整。

⊜ 注意由于该 TTS 是在座席的 Channel 上播放的，因而上述使用 set 设置的 TTS 引擎变量就不起作用了，所以我们在该行单独设置。

- caller_pop：主叫出队。
- consumer_pop：消费者出队。
- bridge-consumer-start：消费者 Channel 开始 bridge。
- bridge-caller-start：主叫 Channel 开始 bridge。
- bridge-consumer-stop：消费者 Channel 停止 bridge。
- bridge-caller-stop：主叫 Channel 停止 bridge。
- consumer_stop：消费者停止。
- channel-consumer-start：消费者 Channel 开始。
- channel-consumer-stop：消费者 Channel 结束。

具体的事件我们就不赘述了。

3. 其他

在 fifo 应用中，通话中的两条腿都是独立建立的，只有这样才能进行桥接（即 bridge）。因而，在普通的通话中靠 a-leg 上的变量去影响 b-leg 的做法（如，使用 export 将变量设置到未来的 b-leg 上）在这里就不适用了。

那么，为了在 b-leg 上设置我们想要的通道变量，可以在向 b-leg 发起呼叫的呼叫字符串上做文章。如下面的设置可以在呼叫 1007 时将主叫号码改为 7777：

```
<member>{origination_call_id_name=7777}user/1007</member>
```

也可以设置与其他业务系统中的座席工号进行关联，如：

```
<member>{my_worker_number=1234}user/1007</member>
```

当然，这里的座席成员除了是本地用户外，也可以是外部系统的号码，如通过某网关能打通的一个手机号：

```
<member>sofia/gateway/pstn/139xxxxxxxx</member>
```

对于 PSTN 网络来讲，它们一般会在呼叫进展阶段返回回铃音或彩铃（对应 SIP 中会收到 183 消息），这会让 FreeSWITCH 认为电话已经接通。因此，为了消除这种影响，一般来说会增加 ignore_early_media 参数，如：

```
<member>{ignore_early_media=true}sofia/gateway/pstn/139xxxxxxxx</member>
```

注意，FreeSWITCH 在拨打 PSTN 网络中的电话之前是无法知道对方是否正常空闲的（也可能是无法接通或被叫忙），它只能按正常的策略去呼出。如果碰巧出现被叫忙或不在服务区之类的情况，PSTN 网络中通常会播放相关的提示音，而不是正常的返回信令消息，这会令接续到下一个座席的时间延长。这是与 PSTN 网络中的电话对接固有的问题。在具体应用时可能还需要注意设置恰当的超时（timeout）等参数，在此我们就不多讲了。

12.2 使用 mod_callcenter 实现呼叫中心应用

上一节讲的 fifo 只是实现了一些先入先出的呼叫队列，与实际的呼叫中心应用相比，还

是显得简单了点。而 mod_callcenter 从名字上看就是一个与呼叫中心应用有关的模块。它实现的 ACD 队列更接近于实际的呼叫中心应用。

对于主叫用户来说，mod_callcenter 采用一种基于积分（score）策略的排队算法。用户在队列中每等待一秒，积分就自动加一[⊖]。在座席端，它实现了一个多级的梯队，允许具有不同技能的座席通过不同的优先级来对用户提供服务。

12.2.1 mod_callcenter 模块简介

要想深入了解 mod_callcenter 模块，就要先安装该模块，并了解其涉及的一些基本概念。

1. mod_callcenter 模块的安装

与 mod_fifo 不同，mod_callcenter 模块并不是默认就编译安装的。因而我们首先要编译安装它。跟其他模块的安装方法类似，可以到 FreeSWITCH 源代码目录中使用以下命令编译安装：

```
# make mod_callcenter-install
```

然后到 FreeSWITCH 控制台上加载该模块：

```
freeswitch> load mod_callcenter
```

如果需要在 FreeSWITCH 启动时自动加载该模块，则可以编辑 conf/autoload_configs/modules.conf.xml，去掉与该模块相关的行的注释，即如下一行：

```
<load module="mod_callcenter"/>
```

2. 涉及的基本概念

mod_callcenter 中有以下基本概念：

- ❑ 队列（Queue）：用于标志一个队列，呼入的电话会在相关的队列中进行排队。
- ❑ 座席（Agent）：座席会事先登录到一个梯队中（后文会讲到），该梯队会与某个队列相关联，当关联的队列中有来话（排队）时，系统会根据一定的策略选择一个空闲的座席为来话客户服务（接听电话）。
- ❑ 策略（Strategy）：队列中座席的分配策略，即当队列中有来话时，如何选择一个空闲的座席。该模块实现了以下几种策略。
  - ring-all：选择所有座席。即让所有座席振铃，哪个先接就选择哪个。
  - long-idel-agent：选择空闲时间最长的座席。
  - round-robin：轮循。
  - top-down：按固定的顺序选择。
  - agent-with-least-talk-time：总是选择通话时间最短的座席；
  - agent-with-fewest-calls：总是选择接电话次数最少的座席；
  - sequentially-by-agent-order：根据梯队和顺序选择；

⊖ 也可以通过人工干预积分的方式来改变主叫用户在队列中的位置。

   - random：随机选择。
- ❑ 座席状态：一个座席有两个状态标志，分别是 Status 和 States。Status 是一个座席逻辑上的状态，它有以下几种取值：
   - Logged Out：退出服务状态。
   - Available：可用状态，可以接电话。
   - Available (On Demand)：一种特殊的可用状态。
   - On Break：座席已登录，但不可以接电话。
- ❑ • States 是跟电话呼叫有关的状态，它有以下几种取值：
   - Idle：空闲。
   - Waiting：等待接受呼叫。
   - Receiving：正在接受呼叫。
   - In a queue call：当前正在一个队列呼叫中。
- ❑ 座席类型：有 callback 和 uuid-standby 两种类型。前者是 onhook 座席，即如果有电话分配到该座席时，FreeSWITCH 将会呼叫座席；而后者是 offhook 座席，即座席需要首先拨入 FreeSWITCH 建立一个呼叫，当有来话时则可以立即与座席桥接起来。
- ❑ 梯队（Tier）：梯队是将队列与座席连接起来的桥梁。另外，它也可以实现多级服务。例如，通过将不同的座席分配到不同的梯队中，可以实现当有电话进来时先连接到普通的座席，有必要时再转到专家座席这样的应用。

12.2.2 座席配置与管理

本小节我们具体介绍座席的配置、管理方法，以帮助大家将前面的所学的知识应用于实践。

1. 静态座席的配置

mod_callcenter 默认的配置文件是 conf/autoload_configs/callcenter.conf.xml，其中配置了一个 support@default 队列，部分配置如下：

```
<queue name="support@default">
  <param name="strategy" value="longest-idle-agent"/>
  <param name="moh-sound" value="$${hold_music}"/>
  ...
</queue>
```

其中，strategy 参数指定了队列的分配方式，它的值（value）longest-idle-agent 说明要优先选择等待时间最长的座席分配。其他的参数我们就不多讲了。

接着，我们来配置一个座席，它的名字是 1005@default，配置如下：

```
<agent name="1005@default" type="callback"
    contact="[call_timeout=10]user/1005"
    status="Available" max-no-answer="3" wrap-up-time="10"
    reject-delay-time="10" busy-delay-time="60" />
```

在该座席的配置中，name 指定座席的名字，它的值为 1005@default，以后都可使用该

名字引用这个座席；type 指定座席的类型；contact 为座席的呼叫字符串，用于在有来话时呼叫该座席；status 为座席的初始状态；max-no-answer 定义座席呼叫失败的次数，如果超过这个值，则将座席状态设为 On Break ；reject-delay-time 为座席拒接后再次选到该座席的最短时长；busy-delay-time 为如果该座席忙，则再次尝试选到该座席的最短时间。

同理我们可以按上面的例子再增加其他座席配置，如 1006@default、1007@default 等。

然后，还需要配置 tier，以将座席与队列关联起来。如下面的配置将三个座席与队列 support@default 关联起来了。

```
<tier agent="1005@default" queue="support@default" level="1" position="1"/>
<tier agent="1006@default" queue="support@default" level="1" position="1"/>
<tier agent="1007@default" queue="support@default" level="1" position="1"/>
```

上述设置完成后，就可以重新加载模块使之生效了：

```
freeswitch> reload mod_callcenter
```

然后，当有电话呼叫（或转移）到 support 时，就可以使用如下的 Dialplan 将电话转到该 callcenter 队列中了：

```
<extension name="Callcenter Example">
    <condition field="destination_number" expression="^support$">
        <action application="answer"/>
        <action application="callcenter" data="support@default"/>
    </condition>
</extension>
```

然后 callcenter 便会根据我们上面设置的规则去呼叫相关的座席。

2. 动态管理队列和座席

mod_callcenter 提供了一个 callcenter_config 的 API 命令，用于管理与该模块相关的各种资源。比如，可以使用如下命令手工将座席签入：

```
freeswitch> callcenter_config agent set status 1005@default 'Available'
```

当然，上述的签入命令用于座席已正确配置的基础上。通过把座席的状态变成 Available，该座席就准备好接听电话了。

下列命令将该座席的状态置为 Logged Out，因此，就不会再有电话分配到该座席了。

```
freeswitch> callcenter_config agent set status 1005@default 'Logged Out'
```

可以用以下命令列出当前所有的座席：

```
freeswitch> callcenter_config agent list
```

当然，如果座席本身没有在配置文件中配置，也可以使用命令添加。比如，下列命令添加一个名字为 1007@default 的座席：

```
freeswitch> callcenter_config agent add 1007@default callback
```

当然，还需要用下列命令给它设置相关的参数，如 contact：

```
freeswitch> callcenter_config agent set contact 1007@default user/1007
```

也可以用以下命令列出当前的队列：

```
freeswitch> callcenter_config queue list
```

或列出当前的梯队：

```
freeswitch> callcenter_config tier list
```

类似的命令还有很多，我们就不一一介绍了。当然，这些命令一般不会仅用手工执行。通过 Event Socket 或 XMLRPC 等接口调用这些命令，则可以做出很友好的界面，也可以很容易集成到其他业务系统中去。

3. offhook 座席

mod_callcenter 也支持 offhook 座席。该模块的 Wiki 页面上有一个示例的 Dialplan 设置，我们把它稍微简化一下：

```
01 <extension>
02   <condition field="destination_number" expression="^(4099)$">
03     <action application="set" data="transfer_after_bridge=4099"/>
04     <action application="sleep" data="300"/>
05     <action application="set"
          data="r=${callcenter_config(agent set uuid ${caller_id_number}@
default '${uuid}')}"/>
06     <action application="set"
          data="r=${callcenter_config(agent set type ${caller_id_number}@
default 'uuid-standby')}"/>
07     <action application="set"
          data="r=${callcenter_config(agent 10 set status ${caller_id_number}@
default 'Available (On Demand)')}"/>
08     <action application="set"
          data="r=${callcenter_config(agent set state ${caller_id_number}@
default 'Waiting')}"/>
09     <action application="set"
          data="cc_warning_tone=tone_stream://%(15 200,0,500,600,700)"/>
10     <action application="answer" />
11     <action application="playback" data="$${hold_music}"/>
12   </condition>
13 </extension>
```

为了便于讲解，我们增加了行号。如果座席呼叫 4099，则会匹配到第 2 行，并进入该 Dialplan；第 3 行设置 transfer_after_bridge 参数，当该座席与来电通话（bridge）完毕后，会再转到 4099 上，重新等待；第 4 行加了一个 sleep，防止执行得太快；第 5 行执行 set，通过 ${API(...)} 的方式调用 callcenter_config API 命令，它将设置本座席的 uuid 参数为当前 Channel 的 UUID；第 6 行将该座席的类型设为 uuid-standby，表示如果有来话就不需要呼叫该座席了，而直接找到前面设置的座席的 uuid 直接将来话和座席的 Channel 桥接（bridge）到一起就可以了；第 7 行设置座席的状态（Status）为可用状态；第 8 行设置座席的呼叫状态（State）；第 9 行通过通道变量 cc_warning_tone 设置一个铃音，当有电话进来时，将向座席播放该铃音；第 10 行应答；第 11 行则播放保持音乐。

至此座席就一直在听音乐，直到有新的电话进来。

除了上面介绍的参数外，mod_callcenter 还有好多配置参数，以及好多可用的命令可以配置不同的队列和运行方式。另外，有一些以 cc_ 开头的通道变量（如上面我们讲到的 cc_warning_tone）也会在某些时候发生作用。关于这些，读者可以多参考该模块的 Wiki 页面，在这里我们就不一一列举了。

mod_callcenter 在排队及电话分配的各个阶段都会产生一些事件，允许第三方的开发者通过 Event Socket 等接口监控队列的行为，并进行各种集成——如可以在 Web 界面上实时显示队列及座席的状态，将状态变化写入数据库供以后产生报表等。当然，要做成一个好的应用前端的开发，工作量还是不小的。

该模块提供的功能与 FreeSWITCH 中其他的模块功能相比，属于更偏上层（应用）的功能。一般来说类似这种功能的应用大家都是使用 Event Socket 等接口在外部进行开发，但若在 FreeSWITCH 内部开发则明显控制力更强一些。当然，不管是在内部开发还是外部开发，对于类似呼叫中心的应用，本模块的一些概念和业务逻辑都是很有参考价值的。

12.3 数据库

FreeSWITCH 内部使用关系型数据库记录一些实时的数据。这些数据有的是临时存储的，有的是持久存储的。关系型数据库很适合描述通信应用中的实体以及关系数据，而且它所支持的各种关系运算也极大简化了 FreeSWITCH 的代码量及开发难度。

12.3.1 默认数据库中有什么？

数据库中到底有什么？很多网友会问这个问题。虽然，一般来讲，用户没有必要直接访问这里的数据库。但是在某些时候直接访问数据库比使用 FreeSWITCH 的命令访问速度更快一些，在合适的场合也不妨合理使用⊖。

1. 核心数据库

我们先从核心看起，看一看默认的数据库中都有什么。

FreeSWITCH 默认使用 SQLite 嵌入式数据库，因而不需要任何配置就能工作。默认的数据库存储位置是 FreeSWITCH 安装目录下的 db 目录，默认的核心数据库名称是 core.db。我们可以先打开该数据库，看看里面到底有什么。

```
dujinfang@seven:/usr/local/freeswitch/db$ sqlite3 core.db
sqlite> .tables
aliases          channels         interfaces       registrations
basic_calls      complete         nat              tasks
calls            detailed_calls   recovery
```

其中，sqlite3 命令用于打开 SQLite 数据库。.tables 命令会显示数据库中所有的表，各表的作用如下：

⊖ 一般来说，进行简单的数据查询或少量的修改算是合理使用。除此之外，有的应用通过在某些数据库表（如用户注册表）上添加触发器，进而触发外部事件，这就显得有些不合适了。

- ❑ aliases：别名表，用于存储命令行别名。
- ❑ basic_calls：一个视图，基于 channels 和 calls 表，提供基本的呼叫信息。
- ❑ calls：呼叫表，在 bridge 的呼叫中，用于关联 Channel 表中的两条腿。
- ❑ channels：存储所有当前的 Channel。
- ❑ complete：存储所有 Tab Complete 数据。
- ❑ detailed_calls：一个视图，基于 Channels 和 Calls，提供详细的呼叫信息。
- ❑ interfaces：存储所有的 Interface。
- ❑ nat：存储当前的 NAT 映射关系。
- ❑ recovery：在使用系统恢复功能功能时，该表存储所有呼叫的详细信息。
- ❑ registrations：存储注册用户的信息。
- ❑ tasks：当前的任务表，如 heartbeat（心跳）、检测 IP 地址变化等。

FreeSWITCH 的 show 命令的大部分内容都是基于这些表的，如 show channels 即查询 Channels 表。当 FreeSWITCH 中有通话时，可以使用如下 SQL 语句进行查询：

```
sqlite> select * from channels;
```

也可以使用 .schema 命令看表的结构，如下列命令就显示了 calls 表的结构：

```
sqlite> .schema calls
CREATE TABLE calls (
   call_uuid  VARCHAR(255),
   call_created  VARCHAR(128),
   call_created_epoch  INTEGER,
   caller_uuid      VARCHAR(256),
   callee_uuid      VARCHAR(256),
   hostname VARCHAR(256)
);
CREATE INDEX calls1 on calls(hostname);
CREATE INDEX callsidx1 on calls (hostname);
CREATE INDEX eeuuindex on calls (callee_uuid);
CREATE INDEX eeuuindex2 on calls (call_uuid);
CREATE INDEX eruuindex on calls (caller_uuid, hostname);
```

2. Sofia 数据库

除核心数据库外，很多外部模块也都使用数据库存储数据。最典型的就是 mod_sofia 模块。

mod_sofia 模块中的每一个 Profile 都使用一个单独的数据库，如我们可以通过下面命令看到每个数据库：

```
dujinfang@seven:/usr/local/freeswitch/db$ ls sofia*
sofia_reg_external.db        sofia_reg_internal-ipv6.db  sofia_reg_internal.db
```

我们打开 sofia_reg_internal.db（即 internal Profile 对应的数据库），看看里面有什么：

```
$ sqlite3 sofia_reg_internal.db
sqlite> .tables
sip_authentication    sip_shared_appearance_dialogs
sip_dialogs           sip_shared_appearance_subscriptions
```

```
sip_presence          sip_subscriptions
sip_registrations
```

其中，SIP 用户的注册信息都存储在 sip_registrations 表中。当然，该表与 core.db 中的 registrations 表中的有些数据是重复的。由于不止 SIP 用户需要注册，其他模块的用户也可能需要注册，核心数据库中存储的是一种统一的注册信息。

其他各表的功能简介如下：

- sip_authentication：SIP 认证相关。
- sip_dialogs：SIP 对话相关。
- sip_presence：SIP 呈现相关。
- sip_shared_appearance_dialogs：SIP SLA 相关，记录当前的 Dialog。
- sip_shared_appearance_subscriptions：SIP SLA 相关，记录订阅信息。
- sip_subscriptions：SIP 订阅相关。

感兴趣的读者可以自己打开看一下，这里就不多介绍了。

3. 其他数据库

其他模块，如我们上面讲过的 mod_fifo、mod_callcenter 等也有自己的数据库。通过直接访问这些数据库，也可以从里面取出有趣的数据，甚至还可以改变模块的运行规则。当然，这类似黑客[㊀]行为，跟前面讲的在数据库上安装触发器差不多，不推荐使用。

关于数据库，我们就不进行更深入的探索了，有兴趣的读者可以参照源代码自行学习。

12.3.2 ODBC

SQLite 在并发量特别大的情况下可能会是一个瓶颈，另外，在一些需要 HA 或负载均衡的场合，也需要一个外部的数据库。FreeSWITCH 支持通过 ODBC [㊁]接口连接外部的数据库。最典型的外部数据库是 PostgreSQL 和 MySQL，也有人成功地连到 Oracle 或 Microsoft SQL Server。

1. 增加 ODBC 数据库支持

为了让 FreeSWITCH 支持 ODBC，在编译前需要安装 unixODBC 的开发包，并安装相应的数据库驱动。在 CentOS、Ubuntu 或 Debian 上的安装 unixODBC 的方式分别如下：

CentOS:

```
yum install unixODBC unixODBC-devel
```

㊀ 这里说的黑客并不含贬义，而是对某个程序或设备进行修改，使之完成原来不可用的功能的人，参见《黑客与画家》(硅谷创业之父 Paul Graham 文集，2011)。自由软件基金会创始人 Richard Stallman 说："出于兴趣而解决某个问题，不管它有没有用，这就是黑客"。

㊁ ODBC（Open DataBase Connectivity，开放数据库互连）是由 Microsoft 提出的数据库接口标准。旨在提供一种标准的 API（应用程序编程接口）方法用于访问数据库管理系统（DBMS）。这些 API 利用 SQL 来完成其大部分任务。ODBC 本身也提供了对 SQL 语言的支持，用户可以直接将 SQL 语句送给 ODBC。ODBC 的设计者们努力使它具有最大的独立性和开放性：与具体的编程语言无关，与具体的数据库系统无关，与具体的操作系统无关。参见 http://zh.wikipedia.org/zh-cn/ODBC。

Ubuntu/Debian:

```
apt-get install unixODBC unixODBC-dev
```

在系统中安装了上述软件包以后，再重新执行 ./configure && make install 的时候，FreeSWITCH 就可以支持 ODBC 数据库了。

2. MySQL

如果使用 MySQL，则需要安装 MySQL 数据库的 ODBC 驱动程序 MyODBC。在不同的系统上可以使用以下命令安装：

Ubuntu:

```
# apt-get install libmyodbc
```

CentOS:

```
# yum install mysql-connector-odbc
```

通过 /etc/odbcinst.ini 可以配置数据库的驱动：

```
[MySQL]
Description         = MySQL driver
Driver              = /usr/lib64/odbc/libmyodbc.so
Setup               = /usr/lib64/odbc/libodbcmyS.so
UsageCount          = 1
FileUsage           = 1
Threading           = 0
```

MySQL 的 ODBC 驱动有一个 _r 版本，如果在你使用的平台上存在这个版本，则必须使用该版本的，如：

```
Driver = /usr/lib64/odbc/libmyodbc_r.so
```

MySQL 的 ODBC 驱动以及客户端库在多线程环境中使用时可能会有问题（如内存泄露），而 FreeSWITCH 是一个典型的多线程系统，因而在生产环境中使用前，需要多做测试，并精心选择一个 ODBC 版本与 MyODBC 版本的有效组合。

接下来需要在 /etc/odbc.ini 中配置 DSN。下面的配置中，DSN（即数据源的名称）为 freeswitch，我们以后可以以这个名字引用它。注意该配置为 64 位平台上的例子，如果使用 32 位的系统，则可能需要将 /usr/lib64 替换为 /usr/lib。另外，里面的 OPTION = 67108864 一行非常重要——FreeSWITCH 会将多条 SQL 语句放到一起执行，以提高效率，如果没有该选项则会出错（MyODBC 3.51.18 及以后的版本才支持该选项）。

```
[freeswitch]
Driver        = /usr/lib64/odbc/libmyodbc.so
SERVER        = localhost
PORT          = 3306
DATABASE      = freeswitch
OPTION        = 67108864
USER          = root
PASSWORD      = password
Threading     = 0
```

3. PostgreSQL

PostgreSQL 的 ODBC 驱动在 Ubuntu 上的安装方法如下：

```
# apt-get install odbc-postgresql
```

/etc/odbcinst.ini 配置如下。注意，下面有两个版本的 PostgreSQL ODBC 驱动。FreeSWITCH 可以使用任意一个。如果你自己在数据库中存储 Unicode 数据的话（如中文），则最好选择 Unicode 版本的。

```
[PostgreSQL ANSI]
Description      = PostgreSQL ODBC driver (ANSI version)
Driver           = /usr/lib/odbc/psqlodbca.so
Setup            = /usr/lib/odbc/libodbcpsqlS.so
Debug            = 0
CommLog          = 0
UsageCount       = 0
Threading        = 0

[PostgreSQL Unicode]
Description      = PostgreSQL ODBC driver (Unicode version)
Driver           = /usr/lib/odbc/psqlodbcw.so
Setup            = /usr/lib/odbc/libodbcpsqlS.so
Debug            = 0
CommLog          = 0
UsageCount       = 0
Threading        = 0
MaxLongVarcharSize=65536
```

/etc/odbc.ini 的配置如下：

```
[freeswitch]
Driver       = /usr/lib64/odbc/psqlodbcw.so
Description = POSTGRESQL
Servername  = localhost
Port         = 5432
Protocol     = 6.4
FetchBufferSize = 99
Username         = postgres
Password         = password
Database         = freeswitch
ReadOnly         = no
Debug            = 1
CommLog          = 1
```

4. 测试 ODBC 连接

ODBC 提供了一个用于连接测试的 isql 命令行工具。只要设置好 ODBC 驱动以及配置文件后，我们这里的测试就与具体数据库类型无关了。命令的使用格式是：

```
isql DSN [UID [PWD]] [options]
```

比如，如果已经将密码写在 odbc.ini 中，我们可以用如下命令进行测试：

```
# isql -v freeswitch
```

也可以明确地提供用户名和密码：

```
# isql -v freeswitch username password
```

其中，-v 选项是可选的，加上它以后可以输出更详细的调试信息。

如果连接成功，将会显示如下界面，然后就可以输入 SQL 语句进行查询了。

```
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
SQL>
```

如果连接失败，则需要查配置文件是否正确，对应的驱动程序是否存在。不同的系统及同一系统不同的版本，驱动程序的路径也不尽相同。

5. 使用 ODBC

当然，也可以把核心数据库放到 ODBC 数据库中，这个可以通过配置 switch.conf.xml 中的如下选项实现：

```
<param name="core-db-dsn" value="odbc://freeswitch:username:password" />
```

其中，dsn 为 ODBC 的 Data Source Name，即数据源的名称，在这里，我们使用了上面配置的名称 freeswitch；username 和 password 分别为连接数据库的用户名和密码。

当然，如果把用户名和密码配置到 odbc.ini 中，则可以直接这样写：

```
<param name="core-db-dsn" value="odbc://freeswitch::" />
```

另外，我们也可以在 mod_sofia 的配置文件（如 internal.xml）或 mod_fifo 的配置文件 fifo.conf.xml 中使用如下参数开启对 ODBC 的支持：

```
<param name="odbc-dsn" value="odbc://freeswitch:username:password"/>
```

配置完上述选择后，重启 FreeSWITCH 或相关模块，就可以使用上述数据库了。在第一次连接一个新的数据库时，FreeSWITCH 会提示找到不相关的表，因而在日志中会出现很多错误。这些错误一般是无害的，FreeSWITCH 会自动创建相关的表，下一次启动就不会报错了。如果不放心的话，可以再重启一次看这些错误是否仍然存在。

12.3.3 使用数据库原生客户端库直接连接数据库

ODBC 在客户端和数据库之间多加了一层中间环节，因而不如直接从客户端连接数据库来得有效。2012 年有人贡献了一些代码，在 FreeSWITCH 内部增加了对 PostgreSQL 的原生支持。我们可以在编译的时候使用如下选项增加这一支持：

```
./configure --enable-core-pgsql-support
```

重新编译安装后，如果一切正常，就可以使用以下方式直接使用原生的客户端库连接数据库了：

```
<param name="odbc-dsn"
    value="pgsql://hostaddr=127.0.0.1
    dbname=freeswitch user=freeswitch password=''
    options='-c client_min_messages=NOTICE'
    application_name='freeswitch'" />
```

在上述配置中，odbc-dsn 参数的值，前面的“ pgsql://”表示是使用 PostgreSQL 原生的客户端库外，后面的表示是一个标准的 PostgreSQL 数据库连接字符串。

现在在 FreeSWITCH 核心中还不能原生地支持 MySQL。FreeSWITCH 官方也不排斥提供对 MySQL 原生支持这个想法，但是到现在为止也没有人愿意贡献这部分代码，而且 PostgreSQL 本身工作得很好，因此在 FreeSWITCH 核心中对原生 MySQL 的支持至今没有进展。

12.4　视频通话

FreeSWITCH 也支持基于 SIP 的视频通话，下面我们来看一看基本的设置，以及视频转码与录像等高级话题。

12.4.1　配置视频通话

FreeSWITCH 默认的配置文件中并没有对视频编解码的相关项，因而默认不支持视频呼叫。如果需要支持视频呼叫，只需要在配置文件中增加相关的视频编解码就可以了。

目前 FreeSWITCH 支持的视频编解码有 H261、H263、H263-1998（H263+)、H263-2000（H263++)、H264、VP8 等。具体应该使用哪种或哪几种编解码需要看 SIP 终端的支持。需要注意的是，与音频编解码不同，FreeSWITCH 中的视频编解码目前仅支持透传，即 FreeSWITCH 仅将通话中一方的视频原样送到另一方去，而不做任何编码转换。这就要求进行视频通信的双方使用一致的编解码[⊖]。

FreeSWITCH 支持的媒体编码默认是在 conf/vars.xml 中定义的，读者可以在该文件中找到类似下面的配置：

```
<X-PRE-PROCESS cmd="set" data="global_codec_prefs=G722,PCMU,PCMA,GSM"/>
<X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=PCMU,PCMA,GSM"/>
```

以上两行分别定义了两个全局变量，它们的字面意思分别是全局的和出局的编解码首选项。假设我们要增加对 H264 和 VP8 编码的支持，则相应的配置如下：

```
<X-PRE-PROCESS cmd="set" data="global_codec_prefs=G722,PCMU,PCMA,GSM,GSM,H264,VP8"/>
<X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=PCMU,PCMA,GSM,GSM,H264,VP8"/>
```

由于上述配置文件实际修改的是全局变量，一般来说需要重启 FreeSWITCH 才能使之

⊖ 目前，比较常用的编解码是 H263、H264 和 VP8。这三种编码都是免费的，而且现在市场上大量的设备都支持 H263 和 H264。但由于 H264 版税的不确定性（现在是免费的，但不知道何时不免费），因而 Google 又大力支持 VP8，Google 在这上面做了大量的工作，力求永远免费。这方面典型的代表就是 Google Chrome 浏览器中的 WebRTC 仅支持 VP8。

生效。

另外，我们在 8.1.3 节也讲了不需要重启 FreeSWITCH 使之生效的一个技巧——如果是在 UNIX 类平台上，可以给 FreeSWITCH 进程发送一个 SIGHUP 信号，让它重新解析全局变量（当然，FreeSWITCH 在收到该信号后还会做其他的事，如执行日志轮转等）。示例如下：

```
# kill -HUP <FreeSWITCH 的进程号>
```

注意，上述 Shell 命令只是保证 FreeSWITCH 正确解析了这些变量。实际上，这些变量的值是在 Sofia Profile 中引用的，因而，还需要在 FreeSWITCH 中执行以下命令来重读 Sofia 的配置，如：

```
freeswitch> sofia profile internal rescan
freeswitch> sofia profile external rescan
```

上面我们重新加载了 internal 和 external 两个 Profile 的配置。当然，也可以用以下命令简单地重新加载 Sofia 模块：

```
freeswitch> reload mod_sofia
```

实际上，我们也可以不修改全局变量，而直接修改 Profile 的配置。如 internal.xml 默认的配置如下：

```
<param name="inbound-codec-prefs" value="$${global_codec_prefs}"/>
<param name="outbound-codec-prefs" value="$${global_codec_prefs}"/>
```

我们可以把视频编码附加到变量引用的后面，具体如下：

```
<param name="inbound-codec-prefs" value="$${global_codec_prefs},H264,VP8"/>
<param name="outbound-codec-prefs" value="$${global_codec_prefs},H264,VP8"/>
```

或直接改成如下形式：

```
<param name="inbound-codec-prefs" value="PCMA,PCMU,H264,VP8"/>
<param name="outbound-codec-prefs" value="PCMA,PCMU,H264,VP8"/>
```

如果出局和入局编解码相同，也可以直接使用同一个参数设置，如：

```
<param name="codec-prefs" value="$${global_codec_prefs}"/>
```

配置完毕后记得通过 rescan 或 reload mod_sofia 命令使之生效。可以使用如下命令查看是否生效（其中 CODECS IN 和 CODECS OUT 分别代表入局和出局时使用的编解码）：

```
freeswitch> sofia status profile internal

=================================================================
Name                internal
CODECS IN           G722,PCMU,PCMA,GSM,G729,H264,VP8
CODECS OUT          PCMU,PCMA,GSM,G729,H264,VP8
```

配置了正确的视频编解码后，就可以在视频话机之间进行视频通话了。

12.4.2　视频录像与回放

前面讲了，FreeSWITCH 支持录音和视频通话，现在我们来讲讲录像。与录音相比，录像要复杂一些。录像数据要按一定的格式存储在文件中，而这些文件格式有好多种。不同的文件格式称为不同的容器（Container），在这些容器中，通常会包含多个音频轨道（Track）和视频轨道，有的还含有同步信息。

目前，处理视频格式最好的开源软件就是 ffmpeg，但是暂时还没有人将它与 FreeSWITCH 集成（这样说也不全对，因为笔者已经在做了）。FreeSWITCH 中实现了一个简单的 mod_fsv ⊖模块，提供 FreeSWITCH 中的录像及回放支持。它不依赖于任何其他的视频处理库，而是自己定义了一种私有的格式，将音频轨道用 L16 编码的数据保存，视频轨道则将整个 RTP 原始包都保存进去。

在默认的 Dialplan 中，也提供了录像与回放的例子。拨打 9193 可以通过 record_fsv App 进行录像，Dialplan 的设置如下：

```
<action application="record_fsv" data="/tmp/testrecord.fsv"/>
```

录像过程中，用户可以看到自己的视频也被原样 echo 了回来。

录制完成后，就可以拨打 9194 播放刚刚录制的录像了，它是使用 play_fsv App 实现的。

```
<action application="play_fsv" data="/tmp/testrecord.fsv"/>
```

上面的录像只能针对单腿的呼叫进行。因为，对于桥接（bridge）的呼叫，需要类似音频的 Media Bug 方式支持录像。而标准的 FreeSWITCH 代码中并不包含视频的 Media Bug 支持。

扩展

笔者在做项目的过程中开发了视频的 Media Bug 支持，并实现了 .fsv 及 .mp4 文件接口，可以支持视频录像。通过使用文件接口，可以直接使用 FreeSWITCH 的 record 和 playback App 进行录像和回放，而不再需要针对每一种视频格式再单独写接口（虽然笔者也实现了 record_mp4 文件接口）。这些代码有望在不久的将来合并到标准的 FreeSWITCH 代码库中。

12.4.3　视频转码

在 FreeSWITCH 支持 WebRTC 之后，人们对视频转码的需求就越来越迫切了。因为在此之前，视频设备一般都支持 H263 或 H264 转码，很容易统一。但有了 WebRTC 以后，由于 Google Chrome 浏览器实现的 WebRTC 仅支持 VP8，与原来的 H26x 的设备进行视频通信就成了问题，而原来的视频设备也不能很快地支持 VP8。当然，Firefox 浏览器也开始支持 WebRTC 了，并支持 H264 格式的视频，但 WebRTC 毕竟是 Google 提出来的标准，因而还是有大量的用户会使用 Chrome。

⊖　参见 http://wiki.freeswitch.org/wiki/Mod_fsv。

通过相应的视频库或硬件的 DSP 芯片，也可以在 FreeSWITCH 中提供视频的转码。在 FreeSWITCH 的作者 Anthony Minessale 的支持下，目前笔者已经开始了这方面的研究，希望在不远的将来会有所突破。

12.5 多人电话会议

FreeSWITCH 支持很强大的多人电话会议（又称音频会议）功能，在会议过程中，可以随时播放声音文件、对会议进行录音，也可以对任意成员进行禁言、禁听等操作。另外，它也支持简单的视频会议功能。下面我们就来一起实验一下会议是怎么工作的，以及如何对会议进行控制的。电话会议功能是在 mod_conference ㊀这个模块中实现的。

12.5.1 音频会议

我们先使用 FreeSWITCH 的默认配置对音频会议来做一个初步的体验。

首先，拿起电话拨打 3000，就进入了一个会议。在会议中，每一个 Channel 都是会议的一个成员（Member），会议中的每个成员都是以 member_id 标志的，它是一个整数值。

由于第一次进入会议时只有一个成员，会议系统会首先播放 "You are the only member in this conference"，即提示该会议中只有你一个成员。

接下来可以用其他电话也呼入 3000 进行电话会议。

1. 使用 DTMF 按键进行控制

在会议中，各成员也可以随时通过 DTMF 按键来进行各种操作。按键对应的操作可以在系统配置文件中配置，默认的控制功能如表 12-1 所示。

表 12-1 电话会议中的功能按键

按钮	说明	按键	说明	按键	说明
1	调低发话音量	2	重置发话音量	3	调高发话音量
4	调低收话音量	5	重置收话音量	6	调高收话音量
7	调低线路能量	8	重置线路能量	9	调高线路能量
*	切换不听不说	0	切换静音	#	挂机

其中，比较常用的是 0，它用于切换本成员静音与否㊁。一般来说，在一个会议中，如果超过 3 个人同时讲话的话，就不容易分辨是谁在说话了；因此，如果某个成员长时间不发言，就可以自发地将自己端的电话静音，以保证会议的质量。另外，有时候有的会议成员的背景环境噪声比较大，如果不发言，也可以使用静音或调低说话音量。

㊀ 参考 https://wiki.freeswitch.org/wiki/Mod_conference。

㊁ 当然，如果使用软电话，也可以将本地的麦克风静音。不过，并不是所有的话机上都有静音按钮，因而，该选项还是很有用的。

2. 配置 Dialplan 将电话路由到会议

mod_conference 提供了一个 conference App，因此，我们可以直接在 Dialplan 中使用它。下面的 Dialplan 是 FreeSWITCH 提供的默认的 Dialplan：

```
<extension name="nb_conferences">
  <condition field="destination_number" expression="^(30\d{2})$">
    <action application="answer"/>
    <action application="conference" data="$1-${domain_name}@default"/>
  </condition>
</extension>
```

可以看出，正则表达式 ^(30\d{2})$ 匹配以 30 开头的 4 位数号码，所以，用户可以呼叫 3000 ~ 3099 之间的号码进入该 Dialplan。电话路由到这里后，将首先执行 answer 动作对来话应答（这步也可以省略，如果后续执行到 conference 的话它也可以自动应答），然后就可以执行到 conference 进入会议了。

其中，conference 的参数 $1-${domain_name}@default 中引用了一些变量，domain_name 一般来说是一个 IP 地址（如 192.168.1.7），如果用户呼叫的是 3000，则上面的参数值就是 3000-192.168.1.7@default。其中，@ 之前的部分为会议的名字，之后的部分为会议的参数。在这里，只有一个参数 default，它是一个会议的 Profile。该 Profile 中定义了一个会议的基本参数和特性。我们先不讨论 Profile 中参数的细节，来接着看下面三个 Dialplan 配置（它们都是在默认 Dialplan 中的，为了节省篇幅，我们省去了其他不关心的内容）：

```
<condition field="destination_number" expression="^(31\d{2})$">
  <action application="conference" data="$1-${domain_name}@wideband"/>

<condition field="destination_number" expression="^(32\d{2})$">
  <action application="conference" data="$1-${domain_name}@ultrawideband"/>

<condition field="destination_number" expression="^(33\d{2})$">
  <action application="conference" data="$1-${domain_name}@cdquality"/>
```

可以把它们与 3000 的那个对比一下，它们除了匹配的被叫号码不同外，进入会议的 Profile 也不同。其中，如果拨打 31 开头的 4 位数电话号码就进入一个宽带（ Wideband，16kHz）的会议；拨打 32 开头的 4 位数电话号码就进入一个超宽带（Ultra Wideband，32kHz）的电话会议；如果拨打 33 开头的 4 位数电话号码就进入一个 CD 音质（CD Quality，48kHz）的电话会议。关于这些 Profile 的区别我们稍后会讲到。

上面我们只讲了 conference App 的 Profile 参数，实际上还可以指定其他参数。conference 的参数总的语法规则是 <conference_name>@<profile>+<pin>+flags{<flag1>|<flag2}。其中，各参数也可以酌情省略。几种有代表性的写法如下：

- 3000：3000 是一个会议名称
- 3000+1234：3000 是会议名称，使用默认的 Profile default，1234 是密码。所有成员在进入会议时都会提示输入密码。
- 3000@default+1234：同上，只是明确指定了 Profle 的名字为 default，当然也可以指定其他 Profile 的名字。

- 3000@default+1234+flags{mute|waste}：在上面的基础上，又增加了 mute 和 waste 参数。
- 3000@default+flags{endconf|moderator}：与上面的类似，只是参数换成了 endconf 和 moderator。

通过上面的语法规则可以看到，会议的名称应该是普通字母加数字组成，不要有“@”、“+”之类的特殊字符。

3. 使用 API 命令控制会议

除了 conference App 外，FreeSWITCH 也提供了一个 conference API 命令用于对会议进行各种控制。在控制台上输入 conference 命令不加任何参数可以显示一些命令帮助信息，如：

```
freeswitch> conference

<conf name> list [delim <string>]|[count]
<conf name> xml_list
<conf name> energy <member_id|all|last|non_moderator> [<newval>]
<conf name> volume_in <member_id|all|last|non_moderator> [<newval>]
<conf name> volume_out <member_id|all|last|non_moderator> [<newval>]
....
```

上面列出了 conference 命令的参数格式，其中，第一个参数是会议的名称；紧接着后面的参数是一个子命令；子命令后面是子命令的参数。不过，上述帮助信息中没列出的是，其中的子命令“list”和“xml_list”前面也可以不需要会议名称，如果没有会议名称就默认列出系统中所有的会议，如：

```
freeswitch> conference list
freeswitch> conference xml_list
```

下面我们按帮助中列出的格式把各子命令简单介绍一下。

注意

我们在这里都以会议名称为 3000 做例子，默认的 Dialplan 生成的会议是类似“3000-IP 地址”这样的会议名称，为了简单起见可以将 Dialplan 中的“-${domain-name}”去掉（在本书的例子中我们大部分都去掉了，这样可以避免每个人的 IP 地址不同造成的差异）。

- list 和 xml_list：列出会议中的成员。
- energy：设置成员的能量值。FreeSWITCH 具有静音检测功能，只有音量超过一定能量才能混音到会议桥中。它的参数首先是一个成员的 ID（member_id），即修改哪个成员的能量，然后是实际能量值。如，ID 为 1 的成员背景噪声比较大，将其的能量提升为 500（默认值为 300）。

```
freeswitch> conference 3000 energy 1 500
```

其中，成员的 ID 除了以整数形式存在的值外，还可以是下列特殊值：

  - all：所有成员

  ➢ last：最后加入的成员
  ➢ non_moderator：非会议主席的成员
- volume_in：调整成员的输入音量，如下面的例子。

```
freeswitch> conference 3000 volume_in 1 1
```

- volume_out：调整成员的输出音量。
- play：向会议中播放声音文件。如向全体成员播放声音的配置如下。

```
freeswitch> conference 3000 play /tmp/test.wav
```

仅向 ID 为 1 的成员播放声音的配置如下。

```
freeswitch> conference 3000 play /tmp/test.wav 1
```

- pause：暂停会议中声音文件的播放。
- file_seek：声音文件的快进、快退。
- say：使用 TTS 功能播放文本。如向会议中播放某一文本的配置如下。

```
freeswitch> conference 3000 say "Hello everyone"
```

- saymember：仅向某个成员播放 TTS。
- stop：停止声音文件的播放。看下面的例子。

```
freeswitch> conference 3000 stop current      # 停止当前正在播放的文件
freeswitch> conference 3000 stop all          # 停止所有正在播放的文件
freeswitch> conference 3000 stop last         # 停止最后播放的文件
```

- dtmf：向某个成员发送 DTMF。
- kick：踢出某个成员。
- hup：挂断某成成员。
- mute：将成员静音（不允许发言）。
- tmute：切换成员的静音状态。
- unmute：取消成员的静音。
- deaf：禁听。
- undeaf：取消禁听。
- relate：使用该功能可以任意控制两个成员间是否可以听到彼此的声音，如下面的例子。

```
freeswitch> confernece 3000 relate 1 2 nohear    # 2 听不到 1
freeswitch> confernece 3000 relate 1 2 nospeak   # 1 听不到 2
freeswitch> confernece 3000 relate 1 2 clear     # 清除建立的关系
```

- lock：锁定会议，别人无法加入。
- unlock：将会议解锁。
- agc：启用自动增益控制（Auto Gain Control）。
- dial：从会议呼出。参数是一个呼叫字符串，如呼叫 1000 加入会议：

```
freeswitch> conference 3000 dial user/1000
```

也可以在呼出时指定主叫名称和号码，如以下命令将在 1000 的话机上显示来电信息为“Seven <7777>”。

```
freeswitch> conference 3000 dial user/1000 Seven 7777
```

- bgdial：同 dial，只是 dial 是阻塞的，bgdial 是异步的，不阻塞当前命令的执行，类似 bgapi。
- transfer：将一个成员转到另一个会议中，如以下命令将成员 1 转到 3001 中。

```
freeswitch> conference 3000 transfer 3001 1
```

- record：对会议录音，如下面的例子。

```
freeswitch> conference 3000 record /tmp/conference.wav
```

- chkrecord：检查录音情况。
- norecord：停止录音。
- pause：暂停录音。
- resume：恢复录音。
- recording：如果在会议的 Profile 中设置了录音模板，则该子命令可以开始和停止录音。如下面的例子。

```
freeswitch> conference 3000 recording start     # 开始
freeswitch> conference 3000 recording stop      # 停止
freeswitch> conference 3000 recording check     # 检查
freeswitch> conference 3000 recording pause     # 暂停
freeswitch> conference 3000 recording resume    # 继续
```

- exit_sound：如果会议中有人退出，所有成员都可以听到一个声音，该命令可以修改这个声音。如下面的例子。

```
freeswitch> conference 3000 exit_sound on                # 启用
freeswitch> conference 3000 exit_sound off               # 禁用
freeswitch> conference 3000 exit_sound none              # 静音
freeswitch> conference 3000 exit_sound file /tmp/a.wav   # 修改成该文件指定的声音
```

- enter_sound：如果会议中有成员进入则播放该声音，用法同上。
- pin：修改会议的密码。
- nopin：取消会议密码。
- get：取得参数的值，这些参数有 max_members（最大成员数）、sound_prefix（声音文件路径）、caller_id_name（主叫名称）、caller_id_number（主叫号码）等。
- set：设置上述变量的值。
- floor：将某成员设为 floor，没什么用。
- vid-floor：设置视频成员的 floor。在视频会议中，所有人都看到持有 floor 的成员。
- clear-vid-floor：取消视频成员的 floor。

4. 配置文件

最后，我们再一起来看一下会议的配置文件，它的默认位置是 conf/autoload_configs/conference.conf.xml。其中，首先配置了一组 DTMF 按键控制功能：

```
<caller-controls>
    <group name="default">
        <control action="mute" digits="0"/>
        <control action="deaf mute" digits="*"/>
        <control action="energy up" digits="9"/>
        <control action="energy equ" digits="8"/>
        <control action="energy dn" digits="7"/>
        <control action="vol talk up" digits="3"/>
        <control action="vol talk zero" digits="2"/>
        <control action="vol talk dn" digits="1"/>
        <control action="vol listen up" digits="6"/>
        <control action="vol listen zero" digits="5"/>
        <control action="vol listen dn" digits="4"/>
        <control action="hangup" digits="#"/>
    </group>
</caller-controls>
```

上述的配置也就是我们在前面看到的按键控制的配置参数，读者可以对比一下看看各参数与实际功能的对应关系。如果需要改变这些按键的功能，建议另外加一个组（group），然后就可以通过组名在后面的 Profile 中引用。

接下来，就可以在配置文件中找到我们前面讲到的那几个 Profile，如：

```
<profile name="default">
<profile name="wideband">
<profile name="ultrawideband">
<profile name="cdquality">
```

可以看到，这些 Profile 之间最主要的区别就是会议的采样率，不同音质的会议采样率不同。我们最常用的 defalt Profile 使用与 PSTN 一致的 8000Hz 的采样率。参数配置如下：

```
<param name="rate" value="8000"/>
```

此外，Profile 的参数中还有一大堆的“-sound”参数，它们配置了在会议中不同时刻播放的声音。如“muted-sound”是在用户被静音时播放给用户的，“unmuted-sound”则相反。有些用户不喜欢在加入会议时听到“You are the only person in this conference”，则可以把“alone-sound”改掉或注释掉。

“comfort-noise”是一个比较有用的参数。在多人会议中，如果没有人发言，则会议中会很安静，大家会误以为会议或者网络出了问题。因此，设置该参数可以产生舒适噪声。该参数的值可以根据需要调整，详情可以参见 12.6.1 节中的 silence_stream 部分。

“auto-record”参数可以设置一个录音模板，当会议室中有至少两个人时，便会自动开始录音。

其他的参数我们就不多介绍了，读者可以自己尝试一下或参考该模块相关的 Wiki 页面。

12.5.2 视频会议

FreeSWITCH 也支持视频会议，下面我们就来看一下 FreeSWITCH 在视频会议方面都有哪些功能。

1. 普通视频会议

按 12.4.1 节中的说明配置好了视频编解码后，就可以拨打 3000 呼入默认的会议进行视频会议了。由于 FreeSWITCH 目前不能对视频进行转码，因而只能是所有人共同看其中一个人的视频图像。那么，具体应该看谁的呢？笔者经常开玩笑说："谁的声音大就看谁的"。当然，这么说不完全准确。实际上，在会议的所有成员中，其中一个成员会有一个 vid_floor 标志，谁持有这个标志，大家就看到谁。这个标志是在会议中由 FreeSWITCH 根据发言者的声音能量值来动态分配的。一般来说，当有多个人轮流说话时，大家就轮流看到正在发言的人，在这种情况下它工作得很好。

当然，视频与音频不同，前者切换没有后者快。而且，在有的视频终端上，可能会出现"花屏"（即屏幕杂乱，出现马赛克之类的）现象。产生这种现象的原因是视频的传输和编码机制造成的。一般来说，视频是以一帧一帧的图像压缩后传送的[㊀]。为了减少传送的数据量，则先传送一个完整的帧（称为关键帧[㊁]），然后后面只传送后面的帧与前面帧的不同的部分（称为非关键帧或预测帧[㊂]）。由于每帧之间图像的差别不是很大，因而可以大大减少传送的数据量。但为了保证画面的清晰流畅以及减少由到丢包等引起的影响，每隔一段时间需要重新发一个关键帧。

所以，在视频会议中发生发言人的视频切换时，如果大家的终端正好错过了新的发言人的终端发出的关键帧，而此时只收到预测帧，在播放时就可能会出现花屏的现象。为了解决这个问题，FreeSWITCH 支持在发生视频切换时主动请求一个关键帧。对于一些旧的 SIP 视频终端，一般支持 picture_fast_update [㊃]请求；对于新的应用，如 WebRTC，则会使用 FIR [㊄]功能请求关键帧。

有时候，在会议中我们可能不希望 FreeSWITCH 自动改变当前看到的成员图像，而是希望手工控制。在 FreeSWITCH 1.4 中，新加入了一个 vid-floor 子命令，可以用它来控制当前哪个成员显示在大家面前。比如，在会议中，以下命令将大家看到的视频切换到 member-

---

㊀ 众所周知，当帧的频率大于一定值的时候，人们的眼睛就可以看到流畅的图像。每秒传送的帧的数量称为帧频，即 FPS（Frame Per Second）。一般来说，电影使用 24FPS、PAL 制式的电视为 25FPS、NTSC 制式的为 29.97FPS，而普通的 SIP 终端一般为 15 或 30FPS。

㊁ 这种完整的帧称为 Intraframe，也称为关键帧，即 Key Frame。在 MPEG 编码（可以认为 H264 是 MPEG 的增强版）中称为 I 帧。

㊂ 这种非关键帧称为 Interframe，也称为预测帧，在 MPEG 编码中称为 P 帧。这种帧需要引用前面的关键帧和预测帧才能还原原始的图像。有的视频系统中也使用双向预测帧，即 MPEG 编码中的 B 帧。但 B 需要前后向的引用，一般不用在实时通信中。VP8 编码中也没有双向预测帧。

㊃ 在 SIP 消息中是通过 INFO 消息实现的，参考 http://tools.ietf.org/html/rfc5168 。

㊄ FIR 的全称为 Full INTRA-frame Request，即完整关键帧请求。它是通过 RTCP 协议实现的，参考 http://tools.ietf.org/html/rfc2032 。

id 为 3 的成员：

```
conference 3000 vid-floor 3 force
```

注意其中的 force 参数，该参数是可选的。如果没有该参数，则只是临时切换到该成员，根据声音大小自动切换还是会出现；如果使用该参数，就固定永远显示这一个成员（当然如果该成员挂机，就又会变成自动选择模式）。

FreeSWITCH 从 1.4 版开始也支持使用 WebRTC 参与会议。不过，由于 FreeSWITCH 不能进行视频转码，因而参与会议的成员必须同使用相同的编码才能相互看到。一般的视频话机都使用 H263 或 H264 编码，而 Chrome 浏览器中实现的 WebRTC 仅支持 VP8 编码。

2. 多画面融屏[1]

另一种比较常见的电话会议是融屏。即将多个成员的图像缩小，再合并到一个图像上，这样，如果会议室中的成员比较少（一般用 4 分屏或 9 分屏，再多了以后图像就很小了）时，就可以看到所有人的图像了。

FreeSWITCH 内部的视频不能进行转码，因而也不支持这种应用。不过，可以通过将视频呼叫转发到其他的 MCU（Multiple Control Unit）上面实现。

笔者在 2013 年 ClueCon 大会上也演示了如何把 FreeSWITCH 做成一个 MCU 的示例[2]。当然，该 Demo 是配合一个硬件板卡实现的，不是标准 FreeSWITCH 的组成部分。不过，在这里，我们也简单介绍一下其实现思路，或许对基于 FreeSWITCH 开发这方面应用的读者有借鉴意义。

由于考虑到软件的视频转码可能会有性能问题，正好有个合作伙伴也提供了一个 DSP[3] 用于测试，笔者就花了一些时间写了个示例程序。

笔者拿到的 DSP 提供 SDK，很快在其提供的示例代码基础上做了一个四分屏融屏程序。实现思路很简单，给视频芯片发过去四路 RTP 视频流，经过融合以后返回来一路 RTP。为简单起见，我们都使用同一种视频编码——H264。实现原理如图 12-1 所示。

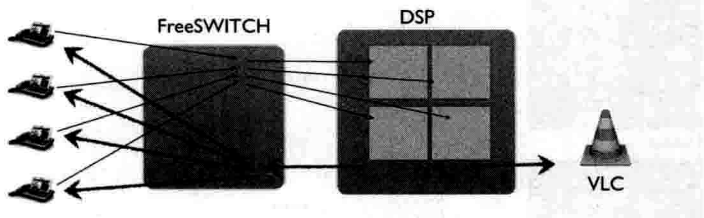


图 12-1 视频融屏原理示意图

其中，我们可以看到四个视频终端的视频流都发送到 FreeSWITCH，我们将它们转发到

---

[1] 注意本节的内容不是标准 FreeSWITCH 具有的功能。在此仅向读者提供一种实现思路。

[2] 部分演讲视频参见 http://v.youku.com/v_show/id_XNjAxMDM4NDQ4.html。

[3] Digital Signal Processor，即数字信号处理器。一般来说，我们服务器上通用的 CPU 不适合处理视频运算，而视频处理一般使用专用的硬件 DSP 或 GPU 实现。

DSP 上，然后控制 DSP 进行融屏，并将结果视频返回给 FreeSWITCH，FreeSWITCH 再将视频发送到各个视频终端上去。当然，为了使例子看起来更灵活，笔者还特意控制 DSP 芯片将融合后的视频也转发了一路到一个 VLC 播放器上播放。

为了实现该应用，笔者修改了 mod_conference，增加了一些代码[⊖]。在会议开始时，每加入一方就将其视频 RTP 流直接转发到 DSP 视频处理芯片上，它会把各路视频融合，并返回一路视频 RTP 流到 FreeSWITCH，在 FreeSWITCH 中启动一个线程专门接收该路视频，并发送到所有视频终端上去。然后所有终端就都能看到所有与会者融合以后的画面，而混音功能仍由 FreeSWITCH 的 mod_conference 提供。

图 12-2 是笔者在芝加哥“彩排”时的截图。其中左上角的视频就是笔者；右上角是一个手机 SIP 客户端发过来的视频；右下角是从一个本地视频文件中播放的视频；左下角是空的，等待其他成员加入。



图 12-2 视频融屏应用截图

3. 视频监控

有了视频会议及融屏的解决方案以后，就可以将视频监控也融入到 FreeSWITCH 中了。为了能与 FreeSWITCH 进行视频通信，最简单的方案就是找一个支持 SIP 协议的摄像机。笔者验证过一款 Grandstream（潮流）高清 IP 摄像机，型号是 GXV3615W。图 12-3 是该摄像机的 SIP 配置页面，可以看出，它跟普通的 SIP 话机配置方式基本上是一样的。



图 12-3 GXV3615W 摄像机 SIP 配置

配置完成后，该摄像机也是以一个 SIP 客户端的方式注册到 FreeSWITCH 上。一般来说，它不会主动发起呼叫，但是我们可以从 FreeSWITCH 中呼叫它。摄像机在接收到呼叫后，会自动应答，所以，我们就可以简单呼叫一个电话号码查看该摄像机覆盖范围内的

⊖ 由于这代码依赖于特定的硬件芯片，因此尚未合并到标准的 FreeSWITCH 代码库中。

图像了。在一些智能家居类的应用中，通过呼叫一个电话号码即可以查看家里的情况。在 FreeSWITCH 中，就这么简单地实现了。

当然，我们也可以将该摄像机加入会议，如（所配置的账号为 1010）：

```
conference 3000 dial user/1010
```

由于该摄像机一般不会说话，所以在 FreeSWITCH 视频会议中如果想看到它的话，就需要使用 vid-floor 功能强制把 floor 设置成它，如：

```
conference 3000 vid-floor 2 force
```

当然，如果使用前面讲到的视频融屏方案，就很容易看到它了。

扩展阅读

在现实生活中，很多摄像机（如马路上的监控）早就安装好了，不可能一下子全部换成 SIP。因此，如何与这些摄像机对接也是一个需要解决的问题。通过研究 FreeSWITCH 的代码，我们给 mod_vlc 模块打了一个补丁[㊀]，让它能支持视频。由于 VLC 库支持相当多的视频格式和协议，不管是一般摄像机使用的 RTSP 协议，还是有些视频编码器输出的 Mpeg2TS 格式，VLC 都能正常播放。因此，打了补丁的 mod_vlc 就可以将这些视频都“收”到 FreeSWITCH 中来，再发到其他任何它支持的视频终端上去。通过这些功能，就可以将不同种类的音、视频客户端用 SIP 联系起来，每个设备也可以分配一个号码，通过这些号码就可以随时“呼叫”这些设备，查看监控现场的情况。而这种思路也与 2011 年国家发布的（2012 年 6 月实施）标准编号为 GB/T 28181-2011 的《安全防范视频监控联网系统信息传输、交换、控制技术要求》[㊁]不谋而合。

总之，FreeSWITCH 作为一个融合平台，将新一代的 SIP 通信与旧时代的各种音频及视频设备融合在一起，可以很容易地做出各种创新、实用的解决方案。

4. RFC4579

RFC4579[㊂]定义了一个基于 SIP 的视频会议控制标准，可以使用标准的视频客户端控制会议里的成员。当会议中的成员有变化时，也可实时通知 SIP 客户端，以便将成员信息显示在客户端的界面上。FreeSWITCH 支持这一特性。它的实现原理是使用了 SIP 中的 SUBSCRIBE/NOTIFY 机制。客户端在启用这一特性时，会向 FreeSWITCH 发起 SUBSCRIBE 请求，订阅与会议相关的资源；FreeSWITCH 在资源的内容发生变化时（如有人入会、出会）便会发送 NOTIFY 消息通知客户端。具体的消息内容是用 XML 定义的。

如果让 FreeSWITCH 支持该特性，需要在会议的 Profile 中开启 RFC4579 的支持，配置如下：

```
<param name="conference-flags" value="rfc-4579"/>
```

㊀ 当然，这部分不是 FreeSWITCH 的标准功能，代码也未开源，在此这种方案和思路仅供读者参考。

㊁ 参见 http://www.sac.gov.cn/gjbzgg/201123/。

㊂ 参见 http://tools.ietf.org/html/rfc4579。

然后，FreeSWITCH 默认配置的 Dialplan 中定义的会议名称与被叫号码不匹配，会导致订阅不匹配的现象，因此，我们把默认的"3000"会议的 Dialplan 中会议名称中的"-${domain_name}"部分去掉，改成如下的样子：

```
<extension name="nb_conferences">
  <condition field="destination_number" expression="^(30\d{2})$">
    <action application="answer" data="is_conference"/>
    <action application="conference" data="$1@default"/>
  </condition>
</extension>
```

其中，可以看出，除了会议名称要与实际的被叫号码匹配外，这一次我们使用的 answer App 多了一个 is_conference 参数，它标志该会议是一个支持 RFC4579 的会议，在 SIP 应答消息"200 OK"中会包含相应的参数，以通知客户端 FreeSWITCH 支持这一特性。当然，在此我们也可以不使用 answer 进行应答，而让来话直接进入会议。在会议中，conference App 会根据 Profile 中定义的参数做出正确的应答选择。

在撰写本书时，笔者已知支持这一特性的唯一客户端是 Mac 版的 Blink[⊖]，因此，在这里我们就用它来进行测试。

在 FreeSWITCH 中经过上面所述的配置，然后用 Blink 注册到 FreeSWITCH 上，呼叫 3000 进入会议，然后用其他客户端也呼入会议，就可以看到如图 12-4 所示的界面，其中，右下角显示了会议室中的两个成员 1000 和 1007。

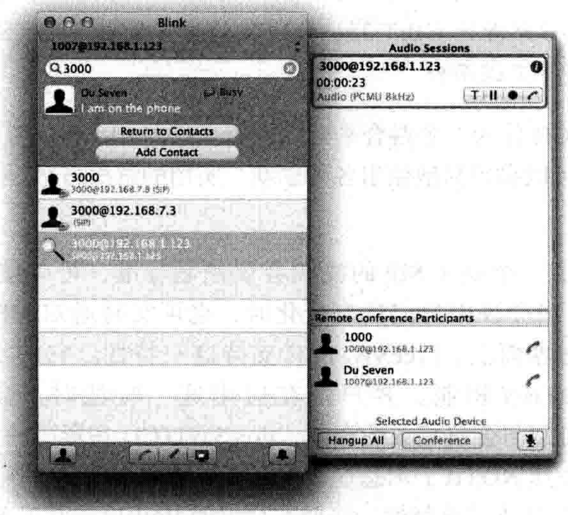


图 12-4 Blink RFC4579 会议界面

当然，如果不使用 Blink 的话，也可以在开发自己的 SIP 客户端时参考 RFC4579，看能否在自己的客户端中增加这些支持。

⊖ 参考 http://icanblink.com/。

12.6 话单

作为一个运营平台，一定要有记录话单（Call Detail Record，CDR）的能力，以便为后续的计费提供数据支持。FreeSWITCH 内部有几个可以记录话单的模块，我们来简单介绍一下。

12.6.1 CSV 格式的话单

mod_cdr_csv[⊖]模块可以记录 CSV（Comma-Separated Values，即以逗号分隔的值）格式的话单，也可以在话单文件中嵌入 SQL 语句，方便后续直接导入到数据库中。该模块是默认加载的。

话单的格式是以模块定义的，默认的配置文件中定义了 sql、example、snom、linksys、asterisk、opencdrrate 等几个模板。默认使用的模板是 example，它是一个 CSV 模板，具体如下：

```
<template name="example">
    "${caller_id_name}","${caller_id_number}",
    "${destination_number}","${context}","${start_stamp}",
    "${answer_stamp}","${end_stamp}","${duration}","${billsec}",
    "${hangup_cause}","${uuid}","${bleg_uuid}","${accountcode}",
    "${read_codec}","${write_codec}"
</template>
```

其中，各字段都是通道变量的引用，在挂机的时候 FreeSWITCH 会将通道变量对应的值替换这里的变量。可以看到，里面包含主叫号码（caller_id_number）、被叫号码（destination_number）、起止时间（start_stamp、end_stamp）、呼叫时长（duration）、计费时长（billsec）等。其中呼叫时长是从呼叫开始到结束的总时间，而计费时长是从应答才开始计算的。

如果使用 sql 模板，就可以直接产生 SQL 语句，模板格式如下（省略部分字段）：

```
<template name="sql">INSERT INTO cdr VALUES ("${caller_id_name}",...
```

当然，也可以参照这些模板写自己的模板。可以在配置文件中使用以下参数配置使用的模板：

```
<param name="default-template" value="example"/>
```

默认情况下，只记录 a-leg 的话单，如果连 b-leg 也记，可以修改如下参数：

```
<param name="legs" value="ab"/>
```

话单文件默认存储在 FreeSWITCH 安装目录的 log/cdr-csv 目录中。其中有 Master.csv 记录了所有的话单，而每个独立的分机也有自己的话单文件。如果只需要记录总的话单，则可以在配置文件中配置如下参数：

```
<param name="master-file-only" value="true"/>
```

话单文件是持续写入的。一般来说，当处理话单时（如将话单导入数据库），需要

⊖ http://wiki.freeswitch.org/wiki/Mod_cdr_csv。

一个“冻结”版本的话单文件，以防止丢失或产生重复话单。为了达到该目的，可以在FreeSWITCH中使用cdr_csv rotate命令让话单文件产生轮转。该命令会将当前话单文件改名为包含有日期时间标志的文件名，如Master.csv.2013-07-29-09-29-41；并且重新打开一个新的话单文件继续写入。这时我们就可以用其他脚本或程序去处理旧的话单文件了。

也可以在Shell中按如下的方式执行该命令，如：

```
# fs_cli -x "cdr_csv rotate"
```

如果配置文件中的rotate-on-hup参数值为true，也可以使用下列命令给FreeSWITCH进程发HUP信号触发日志轮转：

```
# kill -HUP <FreeSWITCH 进程 ID>
```

在UNIX类系统中，还可以把日志轮转加到crontab定时任务中，如：

```
0 * * * * fs_cli -x "cdr_csv rotate"
```

将上述配置加到crontab的配置中，可以每小时将话单轮转一次。当然，也可以在定时任务中直接配置一个处理话单的脚本，在脚本中调用话单轮转命令，并处理产生的话单文件。详细的处理过程我们就不多介绍了，请参考该模块相关的Wiki页面。

12.6.2 直接将话单写入数据库

通过mod_cdr_pg_csv[⊖]模块可以直接在FreeSWITCH中将话单写入PostgreSQL数据库。默认该模块是不自动安装的，因此，在使用前，我们需要手工安装它。

首先，确保系统上安装了PostgreSQL的客户端库。如在Ubuntu Linux上，可以用如下命令安装：

```
# apt-get install libpq-dev libpq5
```

然后就可以按类似其他模块的安装方式来安装该模块了。到FreeSWITCH的源代码目录中，执行如下命令安装：

```
# make mod_cdr_pg_csv-install
```

该模块的配置文件是conf/autoload_configs/cdr_pg_csv.conf.xml，其中有一个参数是配置数据库的连接的，如：

```
<param name="db-info" value="host=127.0.0.1 dbname=freeswitch user=password=
connect_timeout=10" />
```

可以看出，它并没有像其他模块那样使用核心数据库的“pgsql://”字符串方式，而是直接使用的PostgreSQL标准的连接字符串。因为该模块是在核心数据库中加入PostgreSQL之前写的，所以这里的配置格式并不统一。

默认写入的数据库表的名称是“cdr”，也可以使用如下参数修改，如：

```
<param name="db-table" value="freeswitch_cdr"/>
```

⊖ 参考 http://wiki.freeswitch.org/wiki/Mod_cdr_pg_csv。

配置文件中的“schema”标签配置了通道变量与数据库中字段的对应关系，可以根据这里的配置建立数据库表结构。默认的配置如下：

```
<schema>
    <field var="local_ip_v4"/>
    <field var="caller_id_name"/>
    <field var="caller_id_number"/>
    <field var="destination_number"/>
    <field var="context"/>
    <field var="start_stamp"/>
    <field var="answer_stamp"/>
    <field var="end_stamp"/>
    <field var="duration" quote="false"/>
    <field var="billsec" quote="false"/>
    <field var="hangup_cause"/>
    <field var="uuid"/>
    <field var="bleg_uuid"/>
    <field var="accountcode"/>
    <field var="read_codec"/>
    <field var="write_codec"/>
    <!-- <field var="sip_hangup_disposition"/> -->
    <!-- <field var="ani"/> -->
</schema>
```

可以看出，由于 duration 和 billsec 的值是整数值，因此对应的数据库中也应该使用整数（INTEGER）类型的字段，并且我们在 SQL 语句中不需要使用单引号，因此这里加了一个参数 quote="false"。

默认数据库中话单的表结构中的字段名称与这里配置的名称是相同的，如果不相同，可以在 <field> 标签中增加 column 参数，以匹配在数据库中的字段名，如：

```
<field var="local_ip_v4" column="server_ip"/>
```

其他的配置参数与 mod_cdr_csv 中的类似，我们就不多讲了。配置完毕后就可以加载该模块将话单直接写入数据库了。

```
freeswitch> load mod_cdr_pg_csv
```

如果由于某种原因引起写入话单失败，则 FreeSWITCH 就会将这张话单存入一个本地的文件话单池中（默认位置在 log/cdr-pg-csv 目录中），以免丢失话单。在发生这种情况时，可以使用其他的工具将话单池中的话单取出，再导入数据库。

另外，如果你使用的是 MySQL 数据库，则目前 FreeSWITCH 中没有对应的模块，可以考虑使用我们下面介绍的方式实现。

12.6.3 使用 HTTP 服务器接收话单

除了上面介绍的方式外，FreeSWITCH 也支持将话单写入远程的 HTTP 服务器。在那里，你可以使用任何喜欢的编程语言处理话单以及写入任何可能的数据库。有三个模块可以实现该功能：mod_xml_cdr[⊖]、mod_json_cdr 以及 mod_format_cdr。前两者分别产生 XML 和

⊖ http://wiki.freeswitch.org/wiki/Mod_xml_cdr。

JSON 格式的话单，最后一个模块则可以配置产生 XML 或 JSON 格式的话单。

以 mod_xml_cdr 为例，在配置文件中配置如下 url 参数即可以将话单写入指定的 Web 服务器指定的地址上：

```
<param name="url" value="http://localhost/cdr_curl/post.php"/>
```

如果有话单写入，在 Web 服务器端就会收到一个 HTTP Post 请求，请求中带有一个 xml 参数，它的值是一个 XML 文档。然后，在 Web 服务器上就可以取出这个参数，并用 XML 解析器解析里面的内容了。XML 格式的话单内容非常详细（包括电话转接历史等），因而可以很容易在其中获取各种信息。

服务器处理完毕后，应该返回 HTTP“200 OK”消息，然后 FreeSWITCH 就会释放相关的 Channel。如果服务器没有任何反应，FreeSWITCH 就会一直等待，从而产生很多僵尸 Channel，所以一定要保证 Web 服务器能正常响应。当然，Web 技术都发展了几十年了，这种解决方案已经很成熟了。

如果由于某种原因引起写入话单失败，服务器返回非“200 OK”的消息，则 FreeSWITCH 就会将这张话单存入一个本地的 XML 文件中（默认位置在 log/xml_cdr 目录中），以免丢失话单。

12.7 计费

严格来说，计费并不是 FreeSWITCH 要做的工作。FreeSWITCH 只需要忠实地产生正确的话单，实际的计费工作可以由专门的系统去处理。实际的电信运营商也是有专门的计费中心用于话费处理的（因为现在的情况更多的不是以时长（如分钟）计费，而是有各种形式的套餐，在交换机中是不可能知道五花八门的套餐信息的）。

不过，在这里我们还是简单介绍一下 FreeSWITCH 相关的计费方案，以供读者参考。

mod_nibblebill 是一个预付费的计费模块。可以在 FreeSWITCH 源代码目录中使用如下命令配置安装：

```
# make mod_nibblebill-install
```

它需要配合一个外部数据库才能工作，外部数据库可以是通过 ODBC 或内置的 PostgreSQL 原生模式连接。本例中，笔者使用的是 PostgreSQL 数据库。我们先来创建一个数据库表用于存储账户：

```
pgsql> create table accounts (
    id bigserial not null,
    name varchar( 256 ),
    cash double precision not null
);
```

插入一个账户，给我们自己充值 100 元钱：

```
pgsql> insert into accounts (name, cash) values ('Seven', 100);
```

然后，修改模块配置，指向我们的数据库：

```
<param name="odbc-dsn" value="pgsql://hostaddr=127.0.0.1 dbname=freeswitch"/>
```

查询一下数据库中的账户及余额：

```
pgsql> select * from accounts;
 id | name  | cash
----+-------+------
  1 | Seven |  100
(1 row)
```

另外，我们也可以看出我们账户的 id 是“1”，后面我们将根据该 id 计费。

在用户目录中（如 1002.xml），加入以下参数：

```
<variable name="nibble_rate" value="0.01"/>
<variable name="nibble_account" value="1"/>
```

其中，nibble_rate 是每分钟的费率；nibble_account 对应 accounts 表中的 id 字段的值，即对哪个账号进行计费。这样只要该用户打电话，它的费率就是每分钟 1 分钱。

当然，我们也可以改变，账号对应的字段，如，我们可以通过 accounts 表中的 name 字段引用账号：

```
<param name="db_column_account" value="name"/>
```

然后，上述的配置就可以改成这样了：

```
<variable name="nibble_rate" value="0.01"/>
<variable name="nibble_account" value="Seven"/>
```

配置完成后，加载该模块：

```
freeswitch> load mod_nibblebill
```

然后，以后每次分机 1002 打电话，它的账户上就每分钟扣 1 分钱。

当然，我们也可以根据不同的被叫号码设置不同的费率。如我们可以在 Dialplan 中设置，如果是长途呼叫，则每分钟 0.3 元：

```
<extension name="Toll Bill">
  <condition field="destination_number" expression="^(0.*)$">
    <action application="set" data="nibble_rate=0.3"/>
    <action application="bridge" data="sofia/gateway/.....
```

为了保证计费安全，在用户通话过程中该模块将每隔一定时间就从账户上扣一次钱。比如，以下设置实现了每分钟写一次数据库：

```
<param name="global_heartbeat" value="60"/>
```

此外，该模块可以允许设置账号的最低余额、低余额提醒等。通过设置负数的最低余额或者往账号中预充一定的金额，也可以支持后付费的应用。

除了该模块外，还有一个开源的专门用于 FreeSWITCH 计费的产品——vBilling。这款产品笔者没有用过，有兴趣的读者也可以参考官方网站 http://vbilling.org/，在此我们就不再赘述了。

12.8 小结

与普通的 PBX 类应用相比，本章的话题大部分都比较高级。本章我们尽量多地覆盖了一些功能点和特性，仅对某些关键的方面进行了深入的讲解。

首先，我们介绍了使用 mod_fifo 和 mod_callcenter 做呼叫中心应用的实例，以便对想使用 FreeSWITCH 做呼叫中心的读者有一定参考价值。

然后，我们简单介绍了 FreeSWITCH 所使用的数据库的内容，以及如何连接和使用外部的关系数据库等，帮助读者了解数据库的功能和在 FreeSWITCH 中的作用。

接着，我们还详细介绍了 FreeSWITCH 中的多人电话会议功能，并花了很大的篇幅讲了 FreeSWITCH 中的视频通话及视频处理的知识和方法。虽然有些与视频相关的功能和特性不是标准的 FreeSWITCH，但对于使用 FreeSWITCH 进行这方面开发的朋友来说，也有很好借鉴和参考价值的。

最后，我们还讲了在 FreeSWITCH 运营中比较关心的话单和计费的知识。

第 13 章 FreeSWITCH 与 FreeSWITCH 对接

在前面的章节中，我们学习了 FreeSWITCH 的基础知识，并带领大家进行了一些实践，以巩固所学的知识。其实，学好 FreeSWITCH 的关键就是在掌握基础知识，能够用好各种 API 和 App，并配置好 Dialplan。无论如何，前面的章节好像都是我们自己跟自己玩。在真实的世界中，无论是什么产品，是一定要跟别人打交道的，何况我们做的是**通信**。因此，从本章开始，我们逐渐增加一些与其他系统对接的知识。由于我们对 FreeSWITCH 的配置已经比较熟悉，因此，很容易用 FreeSWITCH 来实现或模拟一个“其他系统”。所以，在本章我们先来看一看 FreeSWITCH 跟 FreeSWITCH 是怎么对接的。这样一方面可以复习并巩固以前所学的知识，另一方面也为与其他系统对接打下良好的基础。

明白了多个 FreeSWITCH 之间的连接，就很容易理解如何与其他的设备进行对接了。虽然作为 FreeSWITCH 的粉丝，我想很多读者都会与笔者一样，期待有一天全世界所有的软交换设备都变成 FreeSWITCH。

13.1 在同一台主机上启动多个 FreeSWITCH 实例

为了测试 FreeSWITCH 与 FreeSWITCH 之间的对接，我们需要多个 FreeSWITCH 实例。它们可以运行在同一台主机上，也可以运行在不同的主机上。在此，我们先从在一台主机上启动多个实例讲起。

本章我们会用到多个 FreeSWITCH 进行测试。当然，我们可以用很多台主机进行安装测试，也可以在一台服务器上安装多个虚拟机，但这些操作都费时费力。因此，在这里我们探讨一下多个 FreeSWITCH 实例能否运行在同一台主机上。当然，这对生产环境的 FreeSWITCH 也有指导意义。比如，有时可能需要在一台生产环境的服务器上部署多个不兼容的系统。

13.1.1 背景故事

几年前笔者曾经使用 FreeSWITCH 通过 VoIP 做过在线一对一的网络英语口语教学，由于需要连接 Skype 及 Google Talk。就曾经在同一台主机上做过这样的部署，如图 13-1 所示。

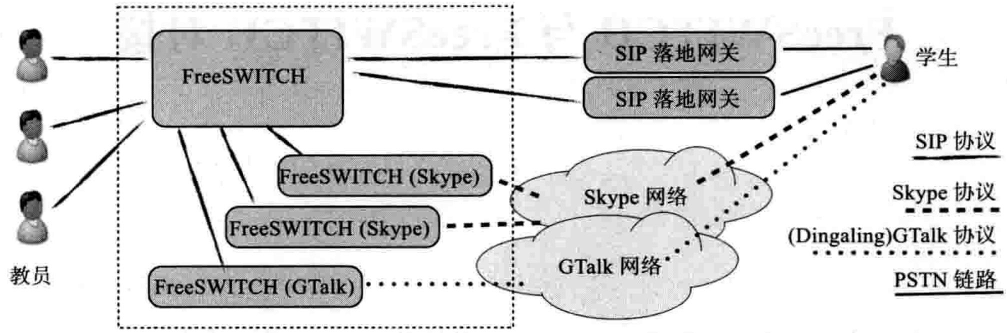


图 13-1 一个网络教学的 FreeSWICH 部署结构

其中，我们的服务器集群（图中虚线部分）在美国。教员也绝大部分在美国，他们的网络条件比较好，因而直接使用客户端通过 SIP 连接 FreeSWITCH。而学生在中国，不能直接使用 SIP，因此，我们直接使用了一些美国运营商的 SIP 落地网关将 SIP 呼叫转换到 PSTN 网络中，再打通学生的手机或固定电话。当然，有时会遇到学生手机信号不好，或者在异地上学习班的情况，此时他们的手机可能会支持高昂的漫游费⊖，为此我们开发了对 Skype 和 Googl Talk（GTalk）的支持，即从 FreeSWITCH 中也可以打到学生的 Skype 或 GTalk 客户端上。

FreeSWITCH 中有支持 Skype 的模块（原先叫 mod_skypiax，后来改名为 mod_skypopen）及 GTalk（mod_dingaling）模块的支持。当然，我们这里主要讲怎么启动多个实例，而具体的模块配置就不讲了。

当时主要考虑到 FreeSWITCH 的 Skype 模块不是很稳定，所以我们把带 Skype 的 FreeSWITCH 启动到另一个实例（对于主的 FreeSWITCH 而言，它们就相当于两个 SIP 到 Skype 的转换网关），这样就避免了由于 Skype 模块崩溃影响到所有业务。当然，为了让 Skype 模块崩溃后也不影响使用 Skype 进行通话的学生，我们后来又启动了另一个带有 Skype 的 FreeSWITCH 实例。这样，正常情况下两个 Skype 网关可以平均分担 Skype 通话，一旦一台崩溃，所有的 Skype 话务就都由另外一台承担。起到了类似 HA（High Availability，高可用）的效果。

后来我们又启动了一个带 mod_dingaling 的 FreeSWITCH 实例，用于与 Google Talk 互通。除了考虑到可能出现的 mod_dingaling 的稳定性问题外，我们主要是为了让呼叫处理更方便。比如，不管是呼叫什么用户，在主 FreeSWITCH 上的呼叫字符串都是类似 sofia/gateway/<gateway-name>/<dest-number> 的形式，从而编程开发方便了许多。

⊖ 比较奇怪的是，我们使用的国际落地长途电话比国内的漫游费都便宜。

13.1.2　练习

这里我们虽不能复现当时的场景，但基本概念是差不多的。我们这里的练习就是在同一机器上启动两个 FreeSWITCH 实例而互相不冲突，甚至可以用各种方式进行互通。

我们都知道 FreeSWITCH 默认的配置文件在 /usr/local/freeswitch/conf 下。这里假设第一个实例已启动并正确运行。

为了启动第二个 FreeSWITCH 实例，首先我们要复制一个新的环境（放到 freeswitch2 目录中，以下的操作都在该目录中）：

```
# mkdir /usr/local/freeswitch2
# cp -R /usr/local/freeswitch/conf  /usr/local/freeswitch2/
# mkdir /usr/local/freeswitch2/log
# mkdir /usr/local/freeswitch2/db
# ln -sf /usr/local/freeswitch/sounds /usr/local/freeswitch2/sounds
```

其中第 1 行创建一个新目录 freeswitch2，第 2 行把旧的配置文件（整个 conf 目录）复制到新目录中，第 4、5 行分别创建 log 和 db 目录，最后一行做个符号链接，确保在新的环境也能正常访问声音文件。

然后，我们需要修改新配置中的一些配置参数以防止端口冲突。第一个要修改的就是 Event Socket 的端口号，它是在 conf/autoload_configs/event_socket.conf.xml 中定义的，找到它并把其中的 8021 改成另一个端口，比方说 9021。

接着修改 conf/vars.xml，把其中的 5060、5080 也改成其他的值，如 7060 和 7080。

如果仅做简单的测试的话，改这两个地方就够了。但如果用于生产环境，还需要修改两个实例的配置文件，将其中的 RTP 端口号的范围改成不冲突的值。FreeSWITCH 中自己维护一个 RTP 端口池，可以在 conf/autoload_config/switch.conf.xml 中修改，默认值如下（修改成不相互冲突的范围就行）：

```
<param name="rtp-start-port" value="16384"/>
<param name="rtp-end-port" value="32768"/>
```

当然如果你还加载了其他的模块，注意要把可能引起冲突的资源都改一下。比如因为笔者要用到 mod_erlang_event 模块，就需要改 autoload_configs/erlang_event.conf.xml 中的 listen-port 和 nodename。

下面我们就可以启动测试了：

```
# cd /usr/local/freeswitch2/
# /usr/local/freeswitch2/bin/freeswitch -conf conf -log log -db db
```

以上命令分别用 -conf、-log、-db 指定新的环境目录。启动完成后将进入控制台。如果想使用 fs_cli 连接该实例的话，则可以打开另外一个终端窗口，使用如下命令连接（还记得前面我们把端口改成 9021 了吧？）：

```
/usr/local/freeswitch2/bin/fs_cli -P 9021
```

找个软电话注册到 7060 端口试试，比如笔者使用 X-Lite 注册时，服务器地址填的是 192.168.1.100:7060。

当然，为了以后启动方便，也可以将上述启动第二个 FreeSWITCH 的命令写到一个简单的脚本里面。例如下面我们使用了 bash 脚本：

```
#!/bin/bash

FSDIR=/usr/local/freeswitch2
cd $FSDIR
$FSDIR/bin/freeswitch -conf conf -log log -db db
```

当然，如果需要启动多个实例，只需要重复本节的步骤，将不同实例的配置文件放到不同的目录中去，将相关的参数修改成不冲突的值就可以了。

13.1.3 进阶

上一节讲到的两个 FreeSWITCH 实例运行的是同一份代码。有时候，你还可能运行两个不同版本的 FreeSWITCH。在这种情况下，在编译的时候就需要指定一个不同的安装目录，比如下面我们指定将新版本的 FreeSWITCH 安装到 /opt/freeswitch 目录中：

```
# ./configure --prefix=/opt/freeswitch
# make && make install
```

安装完毕后，如果执行 /opt/freeswitch/bin/freeswitch 命令，它就默认使用 /opt/freeswitch/conf 目录下面的配置文件，我们也不需要再复制一份配置文件了。

当然，如果需要两个实例同时运行，还是要修改其中一个的相关端口号，以避免冲突。改完以后就可以使用下列命令启动新的 FreeSWITCH 实例了：

```
/opt/freeswitch/bin/freeswitch
```

如果用于生产环境，你可能也想把新的配置加到随操作系统启动的自启动文件里去，这些就留给读者自己练习了。

13.2 FreeSWITCH 与 FreeSWITCH 对接

其实，多机对接的目的就是让不同交换机上的用户能互相打电话。从呼叫上来讲，就是把一个呼叫路由到正确的目的地。在 FreeSWITCH 中，就是设置正确的 Dialplan。在此，我们先来看一下不同的 FreeSWITCH 交换机（不同的主机或不同的实例）之间的对接。

下面我们就带领大家来看几种常见对接方式和组网情况。

13.2.1 双机对接

虽然在 13.1 节我们在同一台机器上启动了两个实例，理论上讲我们可以继续使用上面的例子做下面的练习。但是，由于下面我们还要讲更多的 FreeSWITCH 系统（实例），为了防止混乱，这里还是以每台机器上都有一个 FreeSWITCH 实例来进行讲解，这样我们就可以以不同的 IP 地址代表不同的 FreeSWITCH，讲起来都更容易理解一些。

假设你有两台 FreeSWITCH 主机，分别为 A 和 B，IP 地址分别为 192.168.1.A 和

192.168.1.B。每台机器均使用默认配置，也就是说在每台机器上 1000 ~ 1019 这 20 个号码之间可以互打电话。位于同一机器上的用户称为本地用户，如果需要与其他机器上的用户通信，则其他机器上的用户就称为外地用户。

如果我们需要两台机器之间的用户可以互拨，最需要解决的问题不是技术上如何配置，而是一个逻辑问题，即我们需要先确定一种拨号方案。例如：如果 A 上的 1000 想拨打 B 上的 1000，则 B 上的 1000 相对于 A 上的 1000 来说就是外地用户。就一般的企业 PBX 而言，一般拨打外地用户就需要加一个特殊的号码，比方说 0。这时 0 就称为出局字冠[⊖]。

由上面例子可知，为了完成该实验，我们规定：不管是 A 上的用户还是 B 上的用户，拨打外网用户均需要在实际的电话号码前加拨 0。

在 A 机上，把以下 Dialplan 片断加到 conf/dialplan/default.xml 中（注意在测试时，类似我们在前面讲到的，把下列配置加得“靠前”一点，以防止与其他现有的配置项相冲突）：

```
<extension name="B">
  <condition field="destination_number" expression="^0(.*)$">
    <action application="bridge" data="sofia/external/sip:$1@192.168.1.B:5080"/>
  </condition>
</extension>
```

其中，正则表示式 ^0(.*)$ 表示匹配所有以 0 开头的被叫号码，匹配完成后，括号中的匹配结果会被绑定到变量 $1 中。因此，如果 A 上的用户呼叫 01000，则 $1 的值就是 1000，bridge 是一个 App，它的参数就变成 sofia/external/sip:1000@192.168.1.B:5080，它是一个呼叫字符串。在一个呼叫中，当 FreeSWITCH 执行到这里的 bridge 时，就会从本机的 external Profile（本机的 5080 端口）向 B 的 5080 端口发送 INVITE 呼叫请求。

注意，在上述过程中，被叫号码中的第一个 0 在到达 B 时丢失了。这是系统对接中常用的一个策略，俗称把 0“吃掉”了。

B 在 5080 端口上收到 INVITE 请求后，由于 5080 端口默认走 public Dialplan，所以查找 public.xml，可以找到以下的 Dialplan 配置项：

```
<extension name="public_extensions">
  <condition field="destination_number" expression="^(10[01][0-9])$">
    <action application="transfer" data="$1 XML default"/>
  </condition>
</extension>
```

上述 Dialplan 中的正则表达式 ^(10[01][0-9])$ 会匹配被叫号码 1000，然后执行 transfer，并把来话转到 default Dialplan。呼叫转到 default Dialplan 后，路由规则就跟本地用户的来话一样了，因而最终 B 上的 1000 就会振铃。如果 B 摘机接听，电话就可以接通了。

如果 B 上的用户也要拨打 A 上的用户，那么只需要在 B 上也做类似 A 上的配置就可以了。因为 A 和 B 是完全对称的。

⊖ 以前的交换机又称交换局，因此呼出和呼入就常称为出局和入局。

13.2.2 汇接

理论上来讲，所有的对接模式都可以采用上面的双机对接模式，即上述的对接模式是一切对接的基础。下面我们来考虑一些更复杂的情况。比方说，在此我们假设全世界的交换机都变成 FreeSWITCH 的情况。我们将各 FreeSWITCH 编号为 A、B、C、D、E、F、G……如果 A 上的用户要跟世界上所有的用户都能通话，那么 A 上就需要配置到所有其他主机的路由，这显然是不现实的。

为了解决这一问题，A、B、C、D 开会讨论。D 主动说："我精力比较旺盛，给你们 3 个提供转接服务吧，你们就不用费心了"。这时候，D 就成了一个汇接局，为 A、B、C 之间的通话做转接服务。A、B、C 就称为端局，因为他们只有终端用户（本地用户）。拓扑结构如图 13-2 所示。

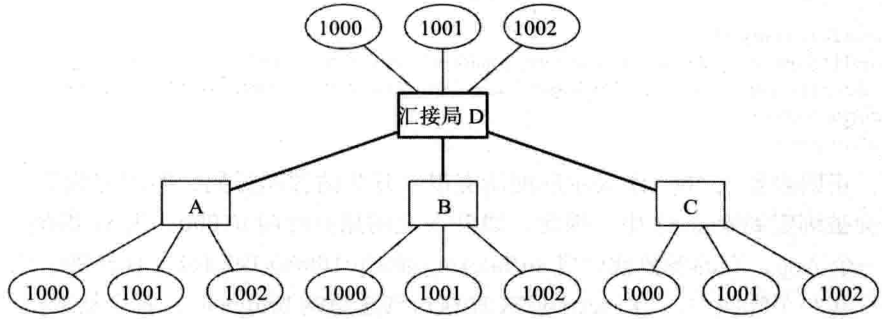


图 13-2 汇接局模式

同时，大家商量了新的拨号规则：本地用户之间的通话拨号规则不变，但是如果拨叫其他外地用户的话，则需要拨打相应的局号（即，如果 A 上的用户呼叫 B 上的 1000，则拨打 B1000），并统一送到 D 进行汇接。我们以 A 为例，它上面的 Dialplan 如下：

```
<extension name="D">
  <condition field="destination_number" expression="^([B-Z].*)$">
    <action application="bridge" data="sofia/external/sip:$1@192.168.1.D:5080"/>
  </condition>
</extension>
```

其中，正则表达式 ^([B-Z].*)$ 表示任何以 B ~ Z 开头的号码（我们假设世界上只有这么多 FreeSWITCH）都送到 D（即 192.168.1.D）的 5080 端口上去。注意，这里并没有"吃掉"第一位号码，因为如果吃掉的话，D 就不知道如何进行下一步路由了。所以，如果 A 上的用户拨打 B1000，在 D 上将收到 B1000。

在 D 上，收到 5080 端口的呼叫请求后，查找 public Dialplan 对来话进行路由。它的 Dialplan 设置如下：

```
<extension name="D">
  <condition field="destination_number" expression="^D(.*)$">
    <action application="transfer" data="$1 XML default"/>
```

```
  </condition>
</extension>
```

上述 Dialplan 说明，如果被叫号码的首位是 D，则说明是一个本地用户，所以“吃掉”首位的“D”，然后把路由转（transfer）到 default Dialplan 进行处理。

对于被叫号码不在本地的用户，则使用下列 Dialplan：

```
<extension name="D">
  <condition field="destination_number" expression="^([A-CE-Z])(.*)$">
    <action application="bridge" data="sofia/external/sip:$2@192.168.1.$1:5080"/>
  </condition>
</extension>
```

其中，正则表示式 ^([A-CE-Z])(.*)$ 匹配所有除 D 以外的 A 到 Z 开头的被号码。我们还是以有人拨打 B1000 为例，匹配成功后，$1 的值为 B，而 $2 的值为 1000，所以，bridge 的参数中呼叫字符串就会变成 sofia/external/sip:1000@192.168.1.B:5080，因而相当于在 D 上“吃掉”了被叫号码中最首位的“B”，并把电话送到 B 的 5080 端口上。

电话到达 B 后，B 上默认的 Dialplan 就可以将电话路由到本地用户上，因而电话接通。这种方式就称为汇接模式。由于 D 即带本地用户，又作为汇接局，因而它是一种混合模式的汇接局。如果 D 上不带用户，而专门做汇接这项工作，就称为一个纯粹的汇接局。

所有局间的通话都是通过 D 的，因而它的压力比较大。在实际应用中，如果 A、B、C 之间的网络直接可达，则可以让 D 仅转发 SIP 信令，而让 RTP 流直接在端局之间传递。在上述配置的 bridge Action 之前增加如下参数可以让 D 工作在 Bypass Media（媒体绕过）模式，仅转发 SIP 信令，而让 RTP 媒体流在端局之间传送：

```
<action application="set" data="bypass_media=true"/>
```

上述参数是在 Dialplan 中设置的，因而仅针对当前通话有效。如果想让 D 对所有通话，不管什么情况都使用 Bypass Media，则可以直接在 Profile（这里我们使用的是 external）中添加如下设置：

```
<param name="inbound-bypass-media" value="true"/>
```

当然，理论上讲，除了 D 之外，其他的端局也可以采用 Bypass Media 技术，让媒体流只在终端用户之间发送，而不经过 FreeSWITCH。但是，一般来说，终端用户可以会位于 NAT 设备的后面，它们之前的媒体流互通并不总是可行的，因而在端局很少这么做。

13.2.3 双归属

即使采用了 Bypass Media，D 的压力还是很大。当然，不一定是电话的压力，更可能是精神上的压力。因为，如果一旦 D 出现了故障，则 A、B、C 之间的用户就都打不通电话了。

为了解决这一问题，它们又找到来了 E，让它做一个备份的汇接局。并且，把 D 和 E 的本地用户都分了一下，放到 A、B、C 上，让 D 和 E 专门做汇接。拓扑结构如图 13-3 所示。

这样就可以把 E 上的路由规则配置得跟 D 上一模一样。但每个端局都同时连接到两个汇接局上。一旦其中一个出现故障，则另一个可以接替工作。这种拓扑方式就称为双归属

(每个端局都归属于两个汇接局)。端局的配置如下（仍以 A 为例，注意，由于“192.168.1”太长了，为了排版方便，我们以“IP.”代替它，读者知道它是一个 IP 地址就行了):

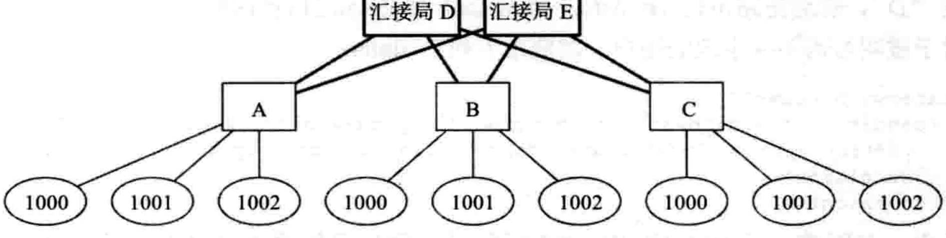


图 13-3 双汇接局双归属网络拓扑

```
<extension name="DE">
  <condition field="destination_number" expression="^([B-Z].*)$">
    <action application="bridge"
      data="sofia/external/sip:$1@IP.D:5080|sofia/external/sip:$1@IP.E:5080"/>
  </condition>
</extension>
```

这里我们修改了呼叫字符串，增加了使用“|”连接起来的两个呼叫字符串。它的意思是，如果第一个呼叫不成功（到 D 的），则使用下一个呼叫字符串（送到 E 上）。这种方式就称为主备用模式。

很容易看出，上述配置虽然解决了 D 的精神压力，但是，它实际的话务压力还是很大的。因为，只要它不出故障，E 就没事干，显然很不公平。所以，实际的双归属汇接局大部分是以话务负荷分担的方式进行的。所谓负荷分担，就是平时在端局将 50% 的通话送到 D 上，50% 通话送到 E 上，让两端的负载差不多。一旦其中一台发生故障，则所有的通话都送到另外一台上。负荷分担的算法有很多，这里我们可以使用如下 Dialplan 实现（以 A 上的出局路由为例）:

```
<extension name="DE">
  <condition field="destination_number" expression="^([B-Z].*[13579])$">
    <action application="bridge"
      data="sofia/external/sip:$1@IP.D:5080|sofia/external/sip:$1@IP.E:5080"/>
  </condition>
</extension>

<extension name="ED">
  <condition field="destination_number" expression="^([B-Z].*[24680])$">
    <action application="bridge"
      data="sofia/external/sip:$1@IP.E:5080|sofia/external/sip:$1@IP.D:5080"/>
  </condition>
</extension>
```

通过上述配置，根据正则表达式 ^([B-Z].*[13579])$，所有被叫号码以奇数结尾的都优先送到汇接局 D 上（如果 D 失败仍然会送到 E)。同理，所有以偶数结尾的被叫号码都会优先送到 E 上，如果失败后则尝试 D。这就实现了一个简单的 50%/50% 负荷分担策略。其他类似的策略可

以使用随机数实现，也可以使用 mod_distributor（FreeSWITCH 中的一个模块，参见 15.5.4 节）中提供的方法实现。当然，实际应用中需要考虑的事情还有很多，这里就不多讲了。

13.2.4 长途局

一般来说，一个地级市的电话网络有两个汇接局就够用了。如果需要在不同的地级市间打长途电话，上面需再建设长途局，以便与其他地级市的长途局互通。有了长途局的网络拓扑结构如图 13-4 所示（我们将另外一个地级市的名称全部以小写字母表示）。

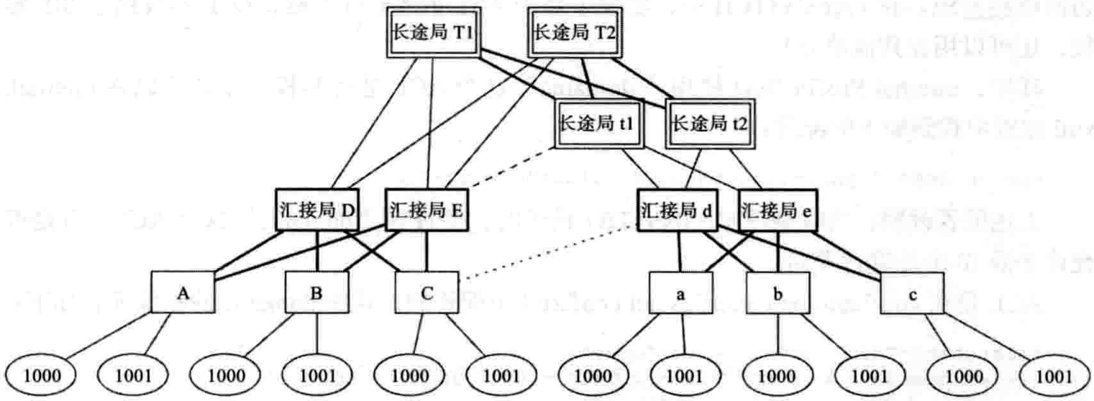


图 13-4 有长途局的网络拓扑

长途局也是采用成对配置的方式，汇接局跟长途局之间也采用双归属。另外，如果某地级市到另外一个地级市的话务量比较高，也可以从汇接局直接向另外一个地市的长途局开中继并设置高速直达路由（如 E 和 t1 之间），它可以避免占用本地长途局的资源。当然，极少数情况下也可能从本地端局直接向另外一个地级市的其他局（如汇接局，C 和 d 之间）开通高速直达中继。

有了长途局，就有了长途区号。我们假设长途局 T1 的长途区号就是 T1，按习惯使用 0 作为国内长途字冠，那么，如果端局 a 上的用户 1000 呼叫端局 A 上的用户 1000 时，它要拨的号码就是 0T1A1000（参见 1.2.1 节）。这样一通电话经过的各交换机的号码分析和路由，最后走的路径是：1000 → a → d（或 e）→ t1（或 t2）→ T1（或 T2）→ D（或 E）→ A → 1000。

具体的配置方式我们就不多讲了。总之，电话网就是这样以树状和网状的结构向外延伸，最终到达世界的每一个角落。电话网内所有的用户也都可以通过一定的拨号规则打通其他的用户的电话。

13.2.5 ACL

在生产环境中，只考虑把电话接通还是不够的，还得考虑安全性。上面的方法只使用 5080 端口从 public Dialplan 做互通，而发送到 5080 端口的 INVITE 是不需要鉴权的，这意味着任何人均可以向它发送 INVITE 从而按你设定的路由规则打电话。这种方式用在端局上

问题可能不大，因为你的 public Dialplan 仅将外面的来话路由到本地用户。但在汇接局模式下，你可能将一个来话再转接到其他局去，那你就需要好好考虑一下安全问题了，因为你肯定不希望全世界的人都通过你的汇接局免费的往各端局打电话。

为了防止这个问题，我们在汇接局上关闭 5080 端口，而让所有来话都送到 5060 端口上（internal Profile）。5060 端口上的来话是需要先鉴权才能路由的。在这种汇接局模式中，一般会使用 IP 地址鉴权的方式。而 IP 地址鉴权就会用到 ACL。

ACL（Access Contorl List）即访问控制列表，它通过一个列表矩阵来控制哪些用户可以访问哪些资源。在 FreeSWITCH 中，实现了基于 ACL 的鉴权（当然，ACL 不仅用于 SIP 鉴权，还可以用在其他地方）。

其中，internal Profile 默认使用“domains”这个 ACL 进行鉴权。读者可以在 internal.xml 配置中找到如下的配置：

```
<param name="apply-inbound-acl" value="domains"/>
```

上述配置说明，当收到呼叫（INVITE）请求时，要查看“domains”这个 ACL，看是否允许来源 IP 地址进行呼叫。

ACL 是在 conf/autoload_configs/acl.conf.xml 中配置的，其中 domains 的默认配置如下：

```
<list name="domains" default="deny">
  <!-- domain= 是一种特殊情况，它会根据用户目录中的配置生成 ACL -->
  <node type="allow" domain="$${domain}"/>
  <!-- 如果你想允许某些 IP 段能通过该 ACL 检查的话，使用 cidr= 的配置方式 -->
  <!-- <node type="allow" cidr="192.168.1.123/32"/> -->
</list>
```

其中，第 1 行说明该列表项的名称是“domains”，可以在其他地方引用，默认（default）的规则是拒绝请求（deny）。其他几行的的含义我们就不在这里介绍了。

如果我们想在 D 上允许来自 A、B、C 的呼叫，就可以把 A、B、C 的地址加到上述配置里面，如可以添加以下配置，以允许（allow）来自这些 IP 的呼叫⊖。

```
<node type="allow" cidr="192.168.1.A/32"/>
<node type="allow" cidr="192.168.1.B/32"/>
<node type="allow" cidr="192.168.1.C/32"/>
```

通过设置上述 ACL，我们就保证了只有授权的用户才能从 D 上进行路由（当然要记得我们关闭了 5080 端口，因此所有呼叫要送到 5060 端口上）。

13.3 FreeSWITCH 作为 PBX

在图 13-4 所示的图网络结构中，假设 A、B、C 一层的都是运营商提供的端局交换机，

⊖ 其中，cidr 为无状态域间路由（Classless Inter Domain Routing）的表示形式，以“/”分开的为 IP 地址和掩码中二进制 1 的位数，32 表示掩码中有 32 个 1，对应的十进制表示即 255.255.255.255，因而它仅表示一台主机；其他常用的掩码位数有 24（255.255.255.0，表示一个 C 类网段）、27（255.255.255.224，表示一个包含 30 台主机 IP 的子网）等。

最底层的 1000、1001 等是端局上的用户。如果再向下延伸，我们可以在端局用户这一层再搭建自己的 PBX，下面再挂分机用户。

13.3.1　普通的 PBX 设置

FreeSWITCH 的配置就是一个全功能的 PBX。假设我们有了 A 上 1000 这个用户账号，我们以前是接了一个 SIP 软电话往外打电话的。现在，我们把它换成 FreeSWITCH（我们称为 F），并且在 FreeSWITCH 上创建 600 ~ 619 这 20 个用户[⊖]，这时候的拓扑结构如图 13-5 所示。

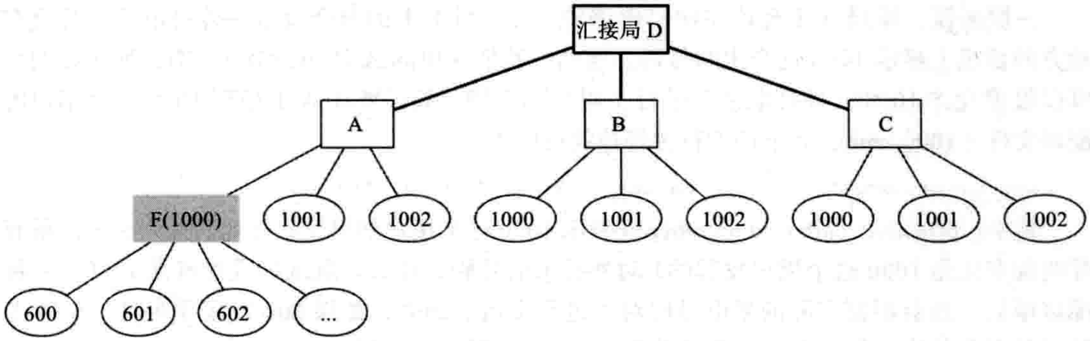


图 13-5　FreeSWITCH 作为 PBX

因为 F 是作为 A 上的一个用户（1000）存在的，所以它只能作为一个普通用户向 A 去注册。对于 A 而言，它就认为 F 是一个普通的 SIP 电话客户端（或软电话）。

为了能使 F 能向 A 注册，我们在 F 上添加一个网关（gateway）指向 A，配置如下[⊖]：

```
<include>
    <gateway name="gw_A">
        <param name="realm" value="192.168.1.A"/>
        <param name="username" value="1000"/>
        <param name="password" value="1234"/>
    </gateway>
</include>
```

添加完上述配置后这样，我们就有了一个名称为 gw_A 的网关。本地的用户 600 ~ 619 就可以通过如下的 Dialplan 拨打外部的电话了（假设 PBX 的出局码为 0）：

```
<extension name="ga_A">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="bridge" data="sofia/gateway/gw_A/$1"/>
    </condition>
</extension>
```

该 Dialplan 的含义就是遇到以 0 开头的被叫号码，“吃掉” 0，然后将电话送到 gw_A 定

⊖ 创建用户我们应该已经很熟悉了。首先可以手工添加这些用户，也可以简单地修改原有的 1000 ~ 1019 这些用户配置文件，改成对应的 600 ~ 619。关于如何添加用户参见 3.3 节及 11.1 节。

⊖ 如何添加网关我们应该已经很熟悉了，可以将下面的内容存放到 conf/sip-profiles/external/gw_a.xml 文件中，并在 FreeSWITCH 中使用“sofia profile external rescan”命令使之生效（参见 3.5 节）。

义的网关上。假设分机 600 想拨打 A 上的 1001，就可以直接拨打 01001，600 的 SIP 客户端将向 F 发送 INVITE（呼叫 01001），F 再向 A 发送 INVITE（1001），对于 A 而言，F 到底是用的 FreeSWITCH 还是用的 SIP 软电话都是一样的。在 1001 上收到呼叫以后，看起来也是从 1000 打过来的（来电显示的号码是 1000）。

如果 A 上的 1001 想呼叫 600，则它必须先呼叫 1000，因为 600 这个号码是无法直达的（如 600 这个人的名片上可能印着“电话：总机 (A)1000 转 600”），然后 F 上会启动一个 IVR，让 1001 输入一个分机号，并为其转接到相应的分机号。

一般来说，端局 A 不允许主叫号码透传，即不管 F 上的哪个分机往外打电话，都会在对方的话机上显示 1000 这个主叫号码。当然，我们这里的 A 是 FreeSWITCH，那么我们就可以设置允许 1000 往外打电话时进行主叫号码透传。只需要在 A 上找到 1000 这个用户的配置文件（1000.xml)，将下面的行注释掉就可以了。

```
<variable name="effective_caller_id_number" value="1000"/>
```

其中，effective_caller_id_number 就表示 1000 这个用户如果发起呼叫时（从 F 来的所有呼叫都变为是 1000 这个用户发起的）对外显示的号码是什么，默认的设置就是 1000。注释掉该项后，就会根据实际的来电号码对外进行发送。那么，如果 600 发起呼叫时，在 A 上看到的实际的来电号码是什么呢？首先，600 发起呼叫时，呼叫到达 F，F 查找 600 的用户配置文件，应该能找到类似的如下的配置，因此 F 将对外发送主叫号码 600。

```
<variable name="effective_caller_id_number" value="600"/>
```

呼叫到达 A 后，它看到 F 送过来的主叫号码是 600，而 1000 这个用户又没有设置 effective_caller_id_number 这个参数（从 F 来的所有通话均认为是从 1000 来的，刚才该参数已经被我们注释掉了）便可以对外显示这个 600，因而 1001 话机上就显示了 600，实现了电话号码的透传。

可以看出，实现电话号码透传本身是很简单的事情。所以它本身不是什么技术问题，而在实际应用中主要是面临一个现实问题，那就是如果所有人都可以做任意透传电话号码，那么所有人都可以任意冒充任何人打电话（试想一下有人冒充 110 或国务院的电话的情况）。

电话号码透传带来的另一个问题就是回呼。在上述情况下，如果 1001 收到 600 来的呼叫，显示的主叫号码是 600，显然这个号码是无法回呼的，即 1001 无法通过这个号码直接呼通 600，因为 600 这个号码理论上讲对于 1001 所在的交换机 A 来说是不存在的。它最多只能呼通到 1000。

为了解决这个问题，我们先把 600 呼出时对外显示的主叫号码变换一下，下面的代码即 Dialplan 将 600 的主叫号码变换为 1000600：

```
<extension name="ga_A">
  <condition field="destination_number" expression="^0(.*)$">
    <action application="set"
      data="effective_caller_id_number=1000${caller_id_number}"/>
    <action application="bridge" data="sofia/gateway/gw_A/$1"/>
  </condition>
</extension>
```

至于为什么这么变换，我们将在下一节给出答案，并给出回呼解决方案。

13.3.2　DID

我们在前面的章节中不止一次讲到 DID 这个概念。在 13.1.1 的方式中，对于 F 而言，号码 1000 就是一个 DID。因为在 A 上的其他用户都可以通过拨打 1000 这个号码打到 F 上，其他交换机如 B、C 上的用户也可以通过拨打 A1000 呼叫到 F 上。当然，从 13.1.1 节我们也看到，其他用户如果想呼叫 F 内部的分机，就只能先拨叫 1000 这个 DID 号码，然后，再通过 IVR，以人工总机或自动总机转接的方式转到对应的分机上。那么，我们能不能直接呼叫到内部的分机呢？毕竟，DID 的真正含义是"对内直接呼叫"，目前看来，我们还未实现这个目标。

要实现这个在技术上当然是可以的。让我们来考虑这样一种拨号方案：对所有 A 上的用户而言（如 1001），如果想直接呼叫 F 上的内部分机 600~619，就需要呼叫 1000 加上这些分机号。如 1001 呼叫 F 上的 600，则需要呼叫 1000600。

我们已经在上一节将 600 对外呼出的主叫号码设置成了可以对外显示 1000600，因此只要能在 A 上设置正确的回呼路由，电话就应该能呼通了。

确定了拨号方案以后，我们再来看在 A 上的实现。为了要让 A 上其他用户打 1000 开头的电话号码都送到 F 上，我们需要在 A 上增加一个 Dialplan 项，于是我们很快想到了如下的 Dialplan：

```
<extension name="F-DID">
  <condition field="destination_number" expression="^(1000.*)$">
    <action application="bridge" data="..."/>
  </condition>
</extension>
```

其中，我们让的正则表达式 ^(1000.*)$ 表示匹配任何以 1000 开头的电话号码。如果匹配到这样的电话号码，则将电话桥接到我们希望的地方。可是到这里，bridge 后面的参数怎么写呢？我们知道，这里我们要给一个呼叫字符串，但是这里的情况不同于我们上面讲到的任何一种情况，呼叫字符串该怎么写呢？

首先，让我们来看一下这里的情况与上述讲的有什么不同。在上面的各种情况中，各 FreeSWITCH 之间的对接都是通过 IP 方式的，即一般来说各 FreeSWITCH 之间都互相知道对方的 IP 地址，也就是说各 FreeSWITCH 的 IP 是相对固定的。而在这里，F 的 IP 地址对 A 来说是不固定的。因为 F 是以动态注册的方式注册到 A 上的，因而只有 F 向 A 注册的时候 A 才能知道 F 的 IP 地址，并且 F 的 IP 是可以不断变化的，每次变化后 F 都会重新向 A 注册它的 IP 地址，以让 A 能找到它。在这种情况下，如果让 A 能找到 F，那么我们应该就能在 A 上动态地获取 F 的 IP 地址了。

实际上，我们已经知道，A 本来就能动态获取 F 的 IP 地址，下面的配置方式我们已经很熟悉了：

```
<action application="bridge" data="user/1000"/>
```

上面的呼叫字符串“user/1000”我们已经讲到很多次了，它的作用就是在A上找到1000这个注册用户注册时的Contact地址（即F的地址），然后向该地址上发送INVITE请求。

但在这里，如果我们还是这样配置的话，很显然不能达到我们的要求。因为使用这种方法获取到的呼叫字符串，在呼叫到达F后，被叫号码（即DID）永远是1000，而我们想要在有人呼叫1000606时把被号码变成1000606。所以，这里我们就需要解决两个问题：①在A上动态能找到F的IP地址；②呼叫到达F后被叫号码变成我们指定的被叫号码。

我们先一步一步来看。当F向A注册时，它提供了自己的Contact地址，我们通过前面的学习已经知道，可以用以下命令在A上查看相关信息（以1000为过滤条件显示注册用户）：

```
freeswitch> sofia status profile internal reg 1000

Registrations:
=================================================================
Total items returned: 0
=================================================================
```

但令人奇怪的是，它并没有显示出1000的注册信息。这是为什么呢？

在回答为什么之前，我们先来试一下以下的命令。在下列命令中我们在显示时去掉了1000这个过滤条件，从中我们可以看到类似如下的输出信息（省略其他不相关的输出）：

```
freeswitch> sofia status profile internal reg

Registrations:
=====================================================================
Call-ID:        d3db5f74-3fb9-4951-ac38-75b419ba4894
User:           1000@192.168.1.F
Contact:        "user" <sip:gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A>
.....
```

从中可以看出，User一行标明我们的确有一条来自192.168.1.F上的用户1000的注册信息，不过该用户的注册信息中的Contact地址是比较奇怪的。由于它里面没有包含1000这样的用户名，所以我们在上面想用1000为过滤条件限制输出结果时失败了。

当然，该Contact字符串来自于F，我们来讲一下F为什么这么做。

F向A注册是以在F中添加一个到A的网关实现，并以“gw_A”这个名字来标志这个网关。F向A注册时使用了这样的Contact字符串。注册完成后，如果A上有电话需要呼叫F时，F就会向A发送INVITE请求。通过上面的注册信息，A向F发的INVITE请求如下：

```
INVITE gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A SIP/2.0
```

这样当F上收到上面的INVITE请求后，就可以知道该呼叫是从gw_A这个网关来的，这样便于知道呼叫的来源。否则的话，F也可以使用像“1000@192.168.1.F:5080”这样的Contact字符串，可是那样的话它会收到类似“INVITE 1000@192.168.1.F:5080”这样的呼入请求，不便于将呼叫来源与F中本身已定义的gw_A网关关联。

继续看上面的INVITE请求。其中有两个“gw_A”。第一个“gw+gw_A”实际上出现

在被叫号码的位置，这时候如果 F 检测到被号号码中有“gw+”以后，就能找到 gw_A 这个网关了。有时候，在 A 是其他 SIP 服务器的情况下，可能会将该字段放入真正的被叫号码，如“INVITE 1000@192.168.1.F...”，这样，F 就会尝试从后面的“gw=gw_A”这个位置（分号后面都相当于参数）找到 gw_A 这个网关名称。当然，如果 A 不发送这些参数的话，F 就无法与系统中配置的网关进行关联了（不过电话仍然可以通）。

F 收到上述的 INVITE 请求后就会在本地的网关 gw_A 中找到真正的被叫号码 1000（从 extension 参数或 username 参数中找）。

实际上，上述的 Contact 地址已经包含了 F 的 IP，我们可以在 A 的命令行上使用下列命令找到它：

```
freeswitch> sofia_contact internal/1000@192.168.1.A
sofia/internal/sip:gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A
```

其中，sofia_contact 是一个 API 命令，它的参数格式为“Profile/User@Domain”。可以看到，我们这里的 Profile 是 internal，User 是 1000，Domain 就是 A 的 IP 地址。

在继续往下研究之前，我们需要先来做一些实验。实验中，我们会用到 echo 和 expand 这两个 API 命令。其中，echo 命令会将字符串原样输出，如：

```
freeswitch> echo test
test

freeswitch> echo $${domain}
$${domain}
```

我们虽然已经知道 A 上的 Domain（即 A 的 IP 地址）是 192.168.1.A，而且它也是相对固定的。但是，在实际应用中还是用变量引用比较方便一些。默认情况下，在 XML 中的配置都是会进行变量替换的，但在命令行上不会。如果要在命令行上进行变量替换，就需要用到一个 expand API 命令：

```
freeswitch> expand echo $${domain}
192.168.1.A
```

可以看到，由于 expand 的作用，$${domain} 被替换成了实际的值。接下来，我们继续进行实验：

```
freeswitch> expand sofia_contact internal/1000@$${domain}
sofia/internal/sip:gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A
```

有了 1000 的 Contact 地址后，我们可以构造如下的命令呼叫它，如：

```
freeswitch> expand echo originate ${sofia_contact(internal/1000@$${domain})} &echo
originate sofia/internal/sip:gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A &echo
```

可以使用“${API 命令 (参数)}”的形式引用一个 API 命令的结果。所以上述的 originate 命令后面的参数需要的呼叫字符串就是用“${sofia_contact()}”动态获取的。当然，上述的命令是 echo 输出后的结果，如果我们去掉 echo，就可以直接执行 originate 呼叫 1000 了：

```
freeswitch> expand originate ${sofia_contact(internal/1000@$${domain}) &echo
```

在 A 上执行上述 originate 命令后，它就会向 F 发送如下的 INVITE 请求：

```
INVITE sip:gw+gw_A@192.168.1.A:5080;transport=udp;gw=gw_A SIP/2.0
To: <sip:gw+gw_A@192.168.1.A:5080;transport=udp;gw=gw_A>
```

然后 F 在收到上述请求后就能找到 gw_A 这个在本地配置的网关，进而找到对应的被叫号码 1000，并进行路由。

如果要将被叫号码改为我们指定的被叫号码（如 1000606），我们只需要想办法改变 INVITE 请求中的被叫号码部分，即将 sip:gw+gw_A@192.168.1.A:5080 变成 sip:1000606@192.168.1.A:5080 即可。为了达到这个目的，我们使用正则表达式替换。其中，FreeSWITCH 提供了一个 regex API 命令可以进行正则表达式替换，它的语法是“regex 原字符串 | 正则表达式 | 替换后的内容”，如：

```
freeswitch> regex sip:gw+gw_A|^sip:gw\+(.*)|sip:1000606
sip:1000606
```

其中，使用“^sip:gw\+(.*)”作为正则表达式匹配就可以将原字符串“sip:gw+gw_A”替换为“sip:1000606”，这也是我们主要需要替换的部分。好了，我们来看一个完整的替换，替换前的字符串为：

```
freeswitch> expand echo originate ${sofia_contact(internal/1005@$${domain})} &echo
originate sofia/internal/sip:gw+gw_A@192.168.1.F:5080;transport=udp;gw=gw_A &echo
```

替换后为：

```
freeswitch> expand echo originate
${regex(${sofia_contact(internal/1005@$${domain})}|^(.*)sip:gw\+xyt@
(.*)|$1sip:1000606@$2)} &echo
originate sofia/internal/sip:1000606@192.168.1.F:5080;transport=udp;gw=gw_A
&echo
```

可以看到，我们得到了需要的呼叫字符串，接下来就可以去掉上面的“echo”进行呼叫测试了。如果在 F 的日志中，看到类似“Processing <0000000000>->1000606 in context public”这样的行就说明我们修改被叫号码成功了。

接下来，可以就可以完成我们的 F-DID Dialplan 了：

```
<extension name="F-DID">
  <condition field="destination_number" expression="^(1000.*)$">
    <action application="bridge"
data="${regex(${sofia_contact(internal/1005@$${domain})}|^(.*)sip:gw\+xyt@
(.*)|%1sip:$1@%2)}"/>
  </condition>
</extension>
```

注意，与在命令行上不同的是，由于 Condition 条件中的正则表达式匹配中的 Capture 结果已经占用了 $1、$2 之类的变量，因而为避免冲突，在 Dialplan 中使用 ${regex()} 之类的表达式时，其 Capture 结果中的变量用 %1、%2 之类表示。

本节给出了一个端局交换机对下面注册上来的 PBX 中的分机用户的一种直接呼叫方案。

虽然该方案不是唯一的解决办法，但它至少是一种有效的解决方案。其他的解决方案原理与此差不多，都是要想办法找到用户注册上来的联系地址，并进行相应的替换。当然，在实际应用中，这种方案是否可行还需要局端交换机 A 与 PBX 交换机 F 相互配合来决定，在此我们就不多介绍了。

13.3.3　使用 PBX 上的网关呼出

下面我们来看另外一种场景。如图 13-6 所示，假设 A 是运行在公网上的 FreeSWITCH，而 F 是运行在私网上的 PBX（也是 FreeSWITCH），F 仍然使用 1000 这个号码向 A 注册，并且 F 上自己带了 600 ~ 619 之间的分机用户。另外，F 上还可以通过另外的一个网关 G 与外界沟通。在现实场景中，G 就可能是一个连接模拟线的模拟网关，该网关一端跟 FS 通过 SIP 相连，另一端则通过模拟电话线连接 PSTN 交换机。由于我们运行在公网上的 A 可能没有对外的中继，因而它上面的用户 1000 ~ 1019 可能也希望通过 F 上的网关 G 对外呼出。

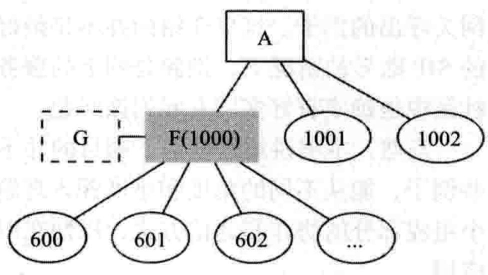


图 13-6　PBX 上带网关的结构图

有了上一节的经验，我们知道实现这个也很简单。首先，就是将 A 上的外呼请求先转发到 F 上。根据上一节的方案，我们可以使用下列 Dialplan 配置来做到：

```
<extension name="F-GW">
  <condition field="destination_number" expression="^(0.*)$">
    <action application="bridge"
data="${regex(${sofia_contact(internal/1005@$${domain})}|^(.*)sip:gw\+xyt@
(.*)|$1sip:${destination_number}@$2)}"/>
  </condition>
</extension>
```

其实该 Dialplan 跟 13.3.2 节的类似。我们只是用正则表达式“^(0.*)$”来匹配以 0 开头的被叫号码，然后通过 bridge 将这种呼叫送到 F 上。在 F 上收到这种呼叫请求后，就可以使用下列的 Dialplan 进行呼出了：

```
<extension name="F-GW">
  <condition field="destination_number" expression="^(0.*)$">
    <action application="bridge" data="sofia/gateway/G/$1"/>
  </condition>
</extension>
```

13.4　小结

在本章，我们首先讲了如何在同一个主机上启动多个 FreeSWITCH 实例。通过这个例子，可以熟悉 FreeSWITCH 的运行方式，也便于以后启动多个 FreeSWITCH 实例进行对接测试。

另外，我们也从双机对接开始，一步步地讲解了大型通信网络的组织及演变，以及FreeSWITCH在各个节点和环节起到的作用。同时也讲解了在FreeSWITCH中是怎么实现这些功能的。以后，如果FreeSWITCH需要与其他系统对接，不管FreeSWITCH是处于哪个位置，总能在本章找到合适的配置方式。读者熟悉了这些配置，就可以以不变应万变，在各种组网环境下都能很快地使用FreeSWITCH实现了。

最后，我们还讲了一个公网的FreeSWITCH服务器通过位于私网的PBX上的中继线或网关呼出的例子。这里介绍的并不是最好的配置，但在开发人员在很难自由获得公网上可用的SIP账号的情况下，做到公网上的服务器与PSTN网络的互联，也不失是一种办法，而且社区中也确实有好多网友问到该问题。

当然，本书讲解这些例子的目的并不是仅仅为了回答网友的提问，而是希望读者通过这些例子，能从不同的角度和维度深入理解FreeSWITCH、理解通信网络，以及通信网络中各个组成部分的协作和通信方式，以便在以后的工作中更有效地解决各种问题，创造更强大的应用。

第 14 章

FreeSWITCH 与其他设备或系统对接

上一章，我们讲了不同的 FreeSWITCH 之间的对接，了解了 FreeSWITCH 世界的配置及组网方式。在本章，我们该走出 FreeSWITCH，去看看外面的世界了。

我们在前面的章节中讲到，FreeSWITCH 支持多种协议，可以与众多的系统进行对接。在这一章，我们就配合一些实例来看一看在 FreeSWITCH 中是怎么配置的。其实，只要读者熟悉了前面章节的内容，配置 FreeSWITCH 与任何系统对接都是没有问题的。但在本章，我们还是通过实例，拓展一些概念和基础知识，并且与读者一起熟悉一些常见的对端设备。读者也可以从这些实例中思考一下对端是怎么实现与我们的 FreeSWITCH 对接的，从而更加深刻地理解一些基本概念和系统的运行机制，以便在以后的应用中做到游刃有余。

另外，本章将要讲到的“其他系统”大部分属于传统的电信设备或系统，因而读者如果对这些传统的概念不熟悉，建议再回过头去复习一下第 1 章和第 2 章中相关的内容。

14.1 使用 Doubango 客户端连接

Doubango [1]是一个不错的开源框架，它跟电信业务走得比较近，主要集中在以 3GPP、TISPAN、Packet Cabel、WiMax、GSMA、RCS-e、IETF 等为标准的 NGN 技术、音视频处理技术、云计算以及 WebRTC 技术等中。该框架使用 ANSI C 编写，具有很好的可移植性。在很多平台上都有基于 Doubango 的客户端实现，如 Windows 上的 Boghe、Mac 上的 iDoubs 以及 Android 系统上的 IMSDroid 等。

但是，我们这里单独介绍 Doubango 并不是因为其强大，而是因为它总是倾向于把简单的东西搞得异常复杂。

我们在第 3 章曾介绍过，最简单的 SIP 注册一共就需要三个选项：服务器地址（Realm）、用户名（Username）和密码（Password），但在 Doubango 中，就没这么简单了。如果你要注册，就得填一大堆的东西，并且格式相当严格和专业，这令好多初学者都摸不着头脑。

[1] 参见 http://doubango.org/。

下面我们以 Mac 平台上的 iDoubs 为例，讲一下它在 FreeSWITCH 上注册的方法。图 14-1 是笔者使用 Mac 版的 iDoubs 注册时的参数设置，在其他平台上可以参考使用。

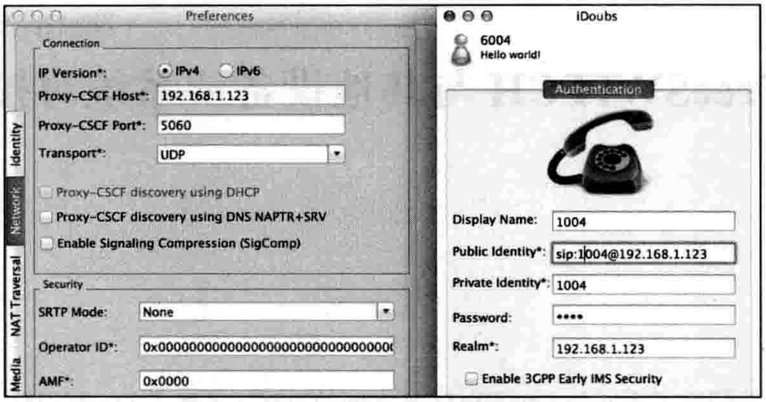


图 14-1 iDoubs 注册设置

要进行注册，首先，要到“Preferences”->“Network”中设置 Proxy-CSCF-Host 和 Proxy-CSCF-Port。这是代理服务器的地址和端口号，相当于其他软件高级设置中的 Outbound Proxy。这个地址的意思是，如果 Doubango 注册时，会将注册的包发到这个地址。而它实际请求域（或简单认为是注册的地址）将是下一步我们将要设置的 Realm 的地址。但由于我们的 FreeSWITCH 只有一个地址，因而这里就填 FreeSWITCH 服务器的 IP 地址。

接下来，要填写注册相关的参数。其中，Display Name 跟其他软件中的一样，是可以随便填的。Public Identity 是 IMS 里的概念，相当于你的公有 SIP 地址，就是说，别人呼叫你的时候要呼叫这个地址。注意这个地址的格式，其中的“sip:”是不能省略的。Private Identity 是你的私有账号，这个在认证的时候要使用，相当于其他软件中的 Username 或 Auth-User（会出现在 SIP 消息的 Authentication 头域中）。Password 就是密码。Realm 是服务器的域（或 IP 地址）。

总之，基于 Doubango 的所有默认应用都倾向于把填写注册信息弄的异常复杂[⊖]。不过，如果顺利过了这一关，大家也能学到不少的知识。尤其是在下一节的对接 IMS 系统时我们还会看到更复杂一些的参数。

14.2 对接 IMS

IMS 是新一代通信网络事实上的标准，它旨在在过渡阶段兼容原有的 PSTN TDM 网络（基于电路交换的网络，在 IMS 中称为 CS 域），并最终将所有的业务都转移到基于包交换的

⊖ 实际上，其他类似的的 SIP 客户端软件也都需要这些信息，但它们都会要求填入最少的信息，如果可能的话，会自动计算这些值，或者有合理的默认值（如 FreeSWITCH 中的网关设置）。Doubango 官方提供的各种应用在这一点上都不是很人性化。

网络（PS 域）中。

目前，国内的运营商都部署了 IMS。IMS 网元众多，系统各网元之间通过高速网络相连。就与交换最相关的核心功能来讲，在 IMS 内部，包含 P-CSCF、I-CSCF、S-CSCF 及 AS 等功能与实体。其中 P-CSCF 位于网络的边缘，接收 SIP 消息，并将 SIP 消息转发到内部的 S-CSCF 及 AS 上。

一般来说，IMS 运行在运营商内部专门的网络上（称为承载网），而在外部是无法接触到运营商的内部网络的。如果要通过互联网与运营商的 IMS 对接，就需要通过 SBC 设备及层层防火墙。SBC 设备横跨运营商的承载网及互联网。所以，一般来说，我们可以通过 SBC 连接到 IMS，进而能够与 PSTN 网络上所有的电话进行通信。所以我们可以把 SBC 设备和 IMS 看成一个 SIP 转 PSTN 的“大”网关。实际上，在实际使用中，大家也不关心是否有 SBC 设备，而只认为运营商有一个“大大”的 SIP 服务器，它可以给我们开放账号。

目前，有些运营商也开始在 IMS 系统上针对一些话务批发商或企业应用放号，因此如何使用 FreeSWITCH 与其对接就是我们需要研究的了。

14.2.1　网关配置

当然，与 IMS 对接最好的方式是使用 SIP 中继方式对接，配置简单使用起来也灵活。但很少有运营商允许你这么做。比较典型的方式还是使用类似电话号码的方式，一个账号一个账号地对接。

下面是一个与云南移动的 IMS 使用账号对接的实例。笔者拿到的是一个试验号码，实际上，对我们来讲，它就是一个 SIP 服务器上的账号。为了与其对接，我们要将 FreeSWITCH 当作一个 SIP 客户端注册到 IMS 上去。在 FreeSWITCH 中，这可以通过添加一个网关实现。下面是一个网关的配置文件：

```
<gateway name="ims">
    <param name="realm" value="ims.yn.chinamobile.com"/>
    <param name="register-proxy" value="211.139.x.x"/>
    <param name="username" value="+86871xxxxxxxx@ims.yn.chinamobile.com"/>
    <param name="password" value="1234"/>
    <param name="from-user" value="+86871xxxxxxxx"/>
    <param name="from-domain" value="ims.yn.chinamobile.com"/>
    <param name="register" value="true"/>
    <param name="outbound-proxy" value="211.139.x.x"/>
</gateway>
```

在网关的配置参数中：

- ims.yn.chinamobile.com 为云南移动内部的域。在公网上，它并不是一个合法的域名。那么我们怎么能注册到该服务器上去呢？实际注册的服务器地址是用 register-proxy 定义的，这里 211.139.x.x 是一个 SBC 的地址。以前，我们在配置网关时都不写这个 register-proxy 参数，它默认就等于 realm。
- username 是 IMS 中的用户账号，它和 password 配合用于鉴权（前者将出现在 SIP 消息中的 Authorization 头域中）。该用户账号中包含了用户所属的域，其用户名部分是

一个 PSTN 网络中的 E164 格式的电话号码，前面几位中的“+86”表示中国的国家代码，“871”是云南昆明的区号，后面的“xxxxxxxx”则是一个本地的电话号码。

- from-user 指定在 SIP 消息中的源用户信息，from-domain 则是指定域，它们会影响 SIP 中的 From 头域。
- register 的值为 true 表示 FreeSWITCH 会向该网关发起注册。
- 前面的信息都是与注册相关的，最后的 outbound-proxy 表示呼叫（即 INVITE 消息）应该发到什么地址，它可以是与注册服务器不同的地址，不过在本例中它与 register-proxy 是相同的。

当然，上述这些参数的值都是在 IMS 系统上配置的。从 IMS 上取得正确值并进行配置后，我们就完成了从 FreeSWITCH 注册到 IMS，这时就可以对外打电话了。

14.2.2 通过 IMS 呼出

完成上一节的配置后，我们可以很快地在 FreeSWITCH 中输入以下命令试一试它是否真的能打到我们的手机上：

```
originate sofia/gateway/ims/0186xxxxxxxx &echo
```

电话接通后，应该能听到自己的声音（回音）。注意，上述账号就相当于一个云南昆明本地的电话号码，因而拨云南以外的手机号需要加“0”。

在 Dialplan 中，进行如下设置就可以通过上面的网关往外打电话了：

```
<extension name="IMS gateway outbound">
    <condition field="destination_number" expression="^(0.*)$">
        <action application="bridge" data="sofia/gateway/ims/$1"/>
    </condition>
</extension>
```

当然，如果运营商给我们开的账号支持号码透传的话，我们就可以透传任何号码了。比如下面的代码，透传了一个美国的 911 出去（注意，最好不要透传 110）：

```
<action application="set" data="effective_caller_id_number=911"/>
<action application="bridge" data="sofia/gateway/ims/$1"/>
```

其实，有意思的是，有的运营商开的账号默认就是支持透传的，这样就会将 FreeSWITCH 内部的分机号（如 1000）透传出去。为了能对所有呼出的电话都显示运营商给我们分配的号码（这里的“xxxxxxxx”），我们可以使用如下 Dialplan 设置：

```
<action application="set" data="effective_caller_id_number=xxxxxxxx"/>
<action application="bridge" data="sofia/gateway/ims/$1"/>
```

也可以使用如下设置：

```
<action application="bridge" data="{origination_caller_id_number=xxxxxxxx}
sofia/gateway/ims/$1"/>
```

14.2.3 通过 IMS 呼入

在我们正确向 IMS 注册后，这种号码也支持呼入。如果有人呼叫该号码（即这里的

“xxxxxxxx”)，IMS 就会给我们的 FreeSWITCH 发 SIP INVITE 消息。我们的 FreeSWITCH 收到该消息后，就可以在 FreeSWITCH 中进行路由了。一般来说，呼入的 DID 是不带区号的本地号码，就是上面网关配置中的“xxxxxxxx”部分，然后，我们就可以设置如下 Dialplan 将来话路由到一个 IVR：

```
<extension name="IMS gateway outbound">
    <condition field="destination_number" expression="^(xxxxxxxx)$">
        <action application="answer" data=""/>
        <action application="ivr" data="demo_ivr"/>
    </condition>
</extension>
```

当然，如果我们很神奇地从运营商那里获得了一个号段（如一个千群或一个万群），也可以把这个号码与我们内部的分机号做成一对一的关系。如，下面 Dialplan 设置可以将来话 DID 的后 4 位与我们内部的分机号一一对应：

```
<extension name="IMS gateway outbound">
    <condition field="destination_number" expression="^xxxx([0-9]{4})$">
        <action application="answer" data=""/>
        <action application="bridge" data="user/$1"/>
    </condition>
</extension>
```

14.2.4 其他问题

与这种 IMS 对接的难点就是调试比较困难。一般来说，也很难找到对端真正懂技术的技术人员配合。所以，在对接遇到问题时，解决的方法基本上就是抓包，看 SIP 消息，然后瞎猜，把所有可能的参数都试一遍。

在一次与 IMS 的对接测试中，发现主叫听不到被叫方的回铃音。经过无数次的探索，终于发现，对方需要一个特殊的 SIP 消息头“P-Early-Media: supported”。该消息头是在 RFC5009[⊖]中定义的。在 FreeSWITCH 的 Dialplan 中，可以使用“sip_h_”开头的通道变量添加扩展的 SIP 消息头，实现添加上述 SIP 消息头的 Dialplan 设置如下：

```
<action application="bridge" data="{sip_h_P-Early-Media=supported}sofia/gateway/
ims/$1"/>
```

一般来讲，SIP 中的“1xx”响应消息是不需要证实的，但在 IMS 系统中，大部分情况下 183 消息也是需要证实的，否则可能无法听到正常的回铃音甚至无法正常接续。与普通的证实消息 ACK 不同的是，对于“1xx”的消息需要使用 PRACK[⊜]（即 Pre-ACK）消息证实。通过在 Profile 中开启如下参数可以让 FreeSWITCH 在收到 183 时发送 PRACK 证实消息：

```
<param name="enable-100rel" value="true"/>
```

通过这些配置，FreeSWITCH 就可以完美地与 IMS 系统对接并进行呼入呼出了。

⊖ 参见 http://tools.ietf.org/html/rfc5009。

⊜ 参见 http://tools.ietf.org/html/rfc3262。

14.3 连接模拟话机和模拟中继线

在实际应用中，不可避免地要连接模拟话机。这里说的模拟电话机就是我们在家里或公司中常见的普通电话机。当然，在没有 FreeSWITCH 之前，我们的电话机就是通过一根普通模拟电话线连接出去的（具体连接到哪儿将在第 14.3.1 中讲到），并通过这条电话线往外打电话。

现在，我们有了 FreeSWITCH，那我们就会想到两个问题：

- 我们能不能把模拟话机也连接到 FreeSWITCH 上？这样模拟话机就可以与我们的 SIP 软电话（或硬电话）通话了。
- 我们的 FreeSWITCH 能不能通过模拟电话线也往外打电话，甚至别人也可以通过与该电话线对应的号码呼叫我们？

答案是肯定的。下面，我们就来看一下这两个问题是怎么解决的。不过，在这之前，我们先来看一些与模拟电话相关的概念。

14.3.1 FXS 和 FXO

普通电话机是模拟的，通过一根模拟线连接到距离最近的电话交换机上。根据所处位置的不同，这个交换机可能是处在运营商机房的交换机，也可能是运营商设在用户小区的一个模块局中的交换机，还可能是本企业自己建设的用户小交换机。近几年，为响应国家光进铜退的号召，有些运营商就直接将光纤拉到用户家里，通过一个“小盒子”（因特网接入设备，称为 IAD）将光纤接口转成普通的以太网口和模拟电话接口，然后再在模拟电话接口上接电话机。总之，我们的普通电话都要通过一根电话线连接到某个设备上。该设备提供一个模拟电话线的接入端口，在技术上，称为 FXS 口。同时，我们的电话机上也有一个模拟线的接口，称为 FXO（在物理上，就是一个 RJ11 接口[1]）。FXS 和 FXO 的连接方式如图 14-2 所示。

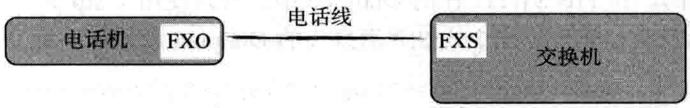


图 14-2 FXS 和 FXO 连接示意图

从常识来讲，我们知道，电话线是带电的，即使家里停电了也可以打电话，如果不小心摸到裸露的电话线也会有“触电的感觉”；而话机本身是不带电的，直接在话机上接上电话线并也不会感觉到有电[2]。由此我们可以联想到，**FXS 口是带电的，而 FXO 口是不带电的**。这也是两者的根本区别。

实际上，FXS 提供的是 –48V 的直流电，主要是为了能感知话机摘挂机的变化、给话机

[1] RJ11 接口最多支持 4 根线，不过模块电话只需要用到其中的 2 根。常见的以太网接口为 RJ45 接口，有 8 根线，不过一般只是用到其中的 4 根。

[2] 有些话机需要安装电池或接变压器电源，这些电源主要是为了支持话机的本地显示屏等功能用的，如支持“来电显示”等。远程交换机上的馈电的功率不足以支持这些功能。

提供拨号音、振铃及其他信号音等。在摘机状态下，电压将降至[1] -7V，而在振铃状态下电压可能增大至 -90V。也可以这样认为，FXS 与 FXO 的区别是：**前者能提供拨号音**。

注意

在实际使用中一定不要将两个 FXS 口用电话线连接在一起，否则两者互相供电，后果是不堪设想的。

14.3.2 拓扑结构

有了上述基本概念以后，我们就可以来看一下要解决我们的问题需要的拓扑结构了。如图 14-3 所示，我们把图 14-2 中的电话线从中间截断，中间放上 FreeSWITCH。FreeSWITCH 提供一个 FXO 接口用于连接原来的交换机；另外提供一个 FXS 接口用于连接原来的模拟电话。此外，其他的 SIP 话机也可以通过以太网 (Ethernet) 口接进来，以实现互相通信。

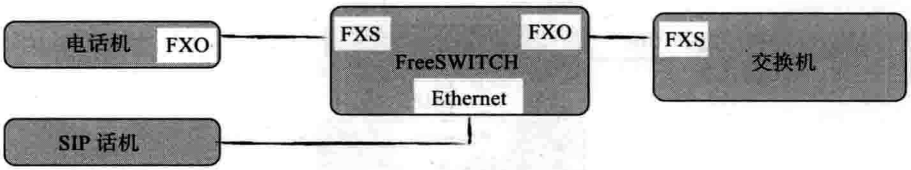


图 14-3 FreeSWITCH 连接 FXS 及 FXO 的拓扑结构

FreeSWITCH 本身是一个软件，因而它是不能提供 FXO 和 FXS 硬件接口的。要解决这一问题，有以下两种方案：

- 在 FreeSWITCH 所在的服务器上安装相关的硬件板卡，该硬件板卡负责提供连接 FXO 及 FXS 的硬件接口。FreeSWITCH 就可以通过相应的驱动程序去控制这些板卡。因而相当于给 FreeSWITCH 增加了 FXO 及 FXS 的接入能力。这类板卡中比较有代表性的有 Sangoma 及 Diguim 生产的模拟和数字板卡等。国内也有许多厂商生产所谓的"Asterisk 兼容卡"，它们有的（如 OpenVox）也能与 FreeSWITCH 搭配使用。
- 通过外部网关[2]来实现。有些厂商针对这一问题专门生产了支持 SIP 到模拟线的转换网关，称为模拟网关。对于 FreeSWITCH 来讲，这种网关就是一个普通的 SIP UA；而对于模拟话机来讲它就是一个电话交换机；对于电话交换机来讲，它就相当于一个普通模拟话机[3]。

---

[1] 在通信设备中一般使用负电压。注意，这里说的是电压的绝对值，虽然 -48 在数学上小于 -7，但对于电压的绝对值来说，从 48 到 7 是下降了。

[2] 一般来说，处于设备的中间，用于连接两个不同网络或不同协议的的设备都称为网关设备。根据网关支持的不同的接口或协议类型以及应用场景又分为多种类型，如模拟网关，E1 数字中继网关等。我们在这里主要讨论 SIP 网关，因此一般来说网关一端的协议是 SIP 的。当然，对于完成同样功能的设备有时也有不同的叫法，如某些型号的路由器设备就支持模拟电话接口，因而也具有网关的功能。另外，有些设备（华为公司生产的终端设备）又称 IAD（Internet Access Device，因特网接入设备），也能完成模拟网关的功能，并可能还支持其他的协议。

[3] 也就是说，带有 FSX 口的设备就相当于一个电话交换机，带有 FXO 口的设备就相当于一个普通模拟话机。

至于如何在以上两种方案之间选择，没有标准的答案。如果有的话，也是具体问题具体分析[⊖]。不过，一般来说，笔者建议：对于模拟线路，选用第二种方案，因为这些网关设备很容易买到，而且坏了可以随时更换，兼容性也比较强；而第一种方案中支持这些板卡的驱动往往需要编译内核等，操作起来比较复杂。后面我们讲到用于数字线路的数字板卡时则可以考虑使用第一种方案，性价比和可控性会高一些。

增加了网关后的拓扑结构如图 14-4 所示。与图 14-3 比较起来，相当于在 FreeSWITCH 中把内置的板卡换成了外置的网关设备。

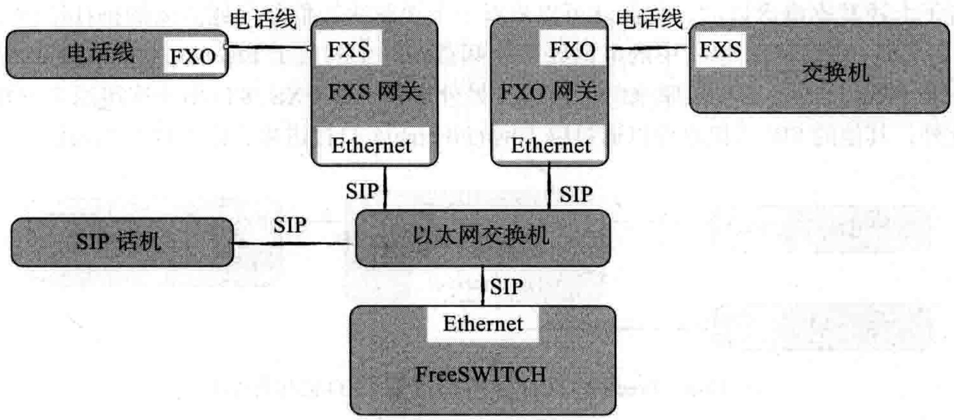


图 14-4 FreeSWITCH 通过网关连接模拟话机及电话线

在实际应用中，有单纯的 FXS 网关及 FXO 网关，端口数从 1 口、2 口到几十口不等。也有的网关是 FXS 和 FXO 混合装在同一设备上的，根据实际需求可以选用不同的网关。

下面我们先来看一下模拟网关的配置和使用。

14.3.3 使用潮流网关连接模拟话机

我们先来看一下如何将话机通过模拟网关连接到 FreeSWITCH 上。潮流网络公司有一款型号为 HT701 的单口模拟网关，它有一个 FXS 口和一个以太网接口，FXS 口用于连接话机，以太接口用于通过以太网连接 FreeSWITCH。如果把它和与之相连的模拟话机看成一体的话，实际上就相当于一个 SIP 话机。也可以说，这款模拟网关能把普通的模拟话机"变"成 SIP 话机。该网关小巧方便，比较适合在桌面上使用。

该网关有一个简单的 Web 配置界面[⊜]，如图 14-5 所示。首先切换到"FXS PORT"配置界面，通过相关配置，可以令其向网关注册。在图中可以看到，它其实跟图 3-13 中亿联

⊖ 常见的答案是 It Depends，即应该根据具体的使用场景来选择。

⊜ 其实使用这类 Web 界面最大的一个问题往往是如何先找到该设备的 IP 地址。不同厂家的设备都有不同的默认地址或者查找 IP 地址的方法，在使用时请参考相关设备说明书或相关资料，在此我们就不多介绍了。下同。

话机的账号配置界面是类似的（因为该网关的功能就相当于一个 SIP 话机）。其中，Primary SIP Server 填入我们 FreeSWITCH 服务器的 IP 地址；Failover SIP Server 是一个备份服务器，用于在 Primary SIP 服务器出现故障的时候自动倒换到 Failover 指定的服务器上，在这里我们不使用，可以不填；SIP User ID 即我们注册的账号，在这里我们使用 FreeSWITCH 默认提供的账号 1000；Authenticate ID 为认证 ID，跟账号一样；Authenticate Password 即密码，填入 1234，Name 为 SIP 中的显示的名字，可以随便起一个；其他的都保留默认配置就可以了。

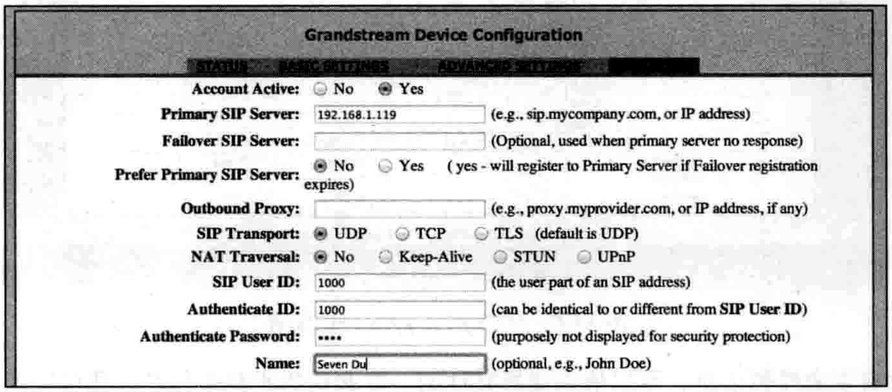


图 14-5 HT701 的配置

正确配置完上述选项后，就可以切换到“Status”页面查看注册状态了。如果注册正常，拿起相连的模拟话机的话筒就可以听到拨号音，然后就可以像正常的 SIP 话机一样打电话了。别人拨打 SIP 账号 1000 时，模拟话机也会振铃。

14.3.4 使用迅时网关连接模拟话机和模拟中继线

上海迅时的网关设备也是国内市场上常见的设备。在本节，我们以迅时 MX8 为例来讲解如何连接模块话机和中继线。MX8 根据情况可以配置 8 个 FXS 口，也可以配置 8 个 FXO 接口，笔者使用的是一款配有 4 个 FXS 接口和 4 个 FXO 接口（简称 4S4O）的网关。

1. 配置 FXS 接口

MX8 网关 FXS 接口的功能和配置跟我们上面讲到的潮流 HT701 差不多。首先，连接 MX8 的 Web 配置界面后，依次选择“基本配置”→“ SIP ”→“注册服务器和代理服务器”菜单项，在配置页面上选择注册方式为“按线路注册”，使用这种注册方式可以单独配置每个 FXS 接口对应的 SIP 账号，相当于 4 个独立的 SIP 话机。

配置好 SIP 服务器的地址后，再转到“线路配置”→“用户线功能”页面，如图 14-6 所示。在线路号码中选择一个 FXS 接口，如 FXS-1；电话号码即我们的 SIP 注册账号，图所示的例子中我们使用 1007，勾选“注册”复选框，然后在密码栏中输入“1234”，提交后，就可以成功注册到我们的 SIP 服务器上了。



图 14-6 MX8 网关 FXS 口的配置

如果有多个模拟话机，可以依次注册其他的。拿起话机听到拨号音，便可以像普通 SIP 电话一样对外呼叫。当然，模拨话机与 SIP 话机毕竟是不同的，这一点体现在拨号方式上尤为明显，要理解这一点需要了解我们下一节将要讲到的拨号规则。

2. 拨号规则

在上一节我们讲到，SIP 话机与普通模拟话机的拨号方式不同。在普通的模拟电话中，拨号时，按下的每一位号码都会实时传到后端的交换机上。因而在打电话时只需要拨完号码等着就行了，交换机在收齐相关号码后会自动帮我们接续。而 SIP 话机是比较“智能”的设备，它能在本地存储号码，在收齐所有号后需要用户按一个“发送”键才能将被叫号码送出去。这一点和模拟电话不同，故往往会很不适应——经常在拨完了号码后忘了按发送键，瞎等半天没有任何反应。不过，这一点倒是跟手机有些类似，手机也是在收齐了号码后需要按“发送”键才能拨出的。

这里我们还是讨论模拟话机。在用户摘机后，相连的交换机就开始向话机播放拨号音并启动收号程序。一般来说，电话号码是有规律的。比方说 110，交换机知道它是一个短号码，收到三位 110 后就不会等待下面的拨号了，保证比较快速地接续。另外大家熟悉的手机号码也都是定长的，所以交换机在收齐足够的位数后就会快速接续。

现在，我们将模拟话机接到 MX8 网关上，因而该网关就需要具备收号功能。为了能保证快速地接续，它定义了一些拨号规则。如果所拨的号码匹配该拨号规则，就会将号码立即送出去（即向 SIP 服务器发送 INVITE 消息）。笔者使用的网关默认的拨号规则如下：

```
01[3,5,8]xxxxxxxxx
010xxxxxxxx
02xxxxxxxxx
```

```
0[3-9]xxxxxxxxxx
120
11[0,2-9]
111xx
123xx
95xxx
100xx
1[3,5,8]xxxxxxxxx
[2-3,5-7]xxxxxxx
…（后面略）
```

可以看出，上述规则中第 1 行是匹配常见的手机号，第 2 行匹配北京的固定电话号码，其他的依此类推。我们在这里不介绍这些规则的具体含义，读者如果在使用时可以自行参考网关设备的参考手册。在这里需要注意的是，这些默认的拨号规则不一定适合你的需要。比如，如果你想拨打内网的一个分机号 1200，由于在规则匹配时匹配到第 5 行的 120 就终止了，因而 INVITE 消息中的被叫号码就只是 120，只要有这条规则存在，1200 就永远也拨不出去⊖。

当然，如果你不理解这些的具体含义又一时找不到手册的话也不要紧，可以尝试把这些规则全删掉，而在拨号时拨完所有号码后在后面再加上一个“#”号，MX8 网关收到“#”号后就会立即将前面收到的号码送出，而不包含最后的“#”号（当然，如果被叫号码中要求包含“#”号的情况下还是可能会有问题，那时候就真需要参考设备手册了）。

3. 配置 FXO 接口连接外线

FXO 接口可以连接运营商提供的电话线（我们称为外线）与外界通话。在 MX8 中，连接到 FXO 接口的电话线称为中继线（见 2.1.2 节中的模拟中继线部分）。切换到“线路配置”→“中继线功能”可以看到如图 14-7 所示的界面。在这里，在线路号码中选择一个端口，如 FXO-1；外线号码填入运营商给我们分配的电话号码（实际上在这里可以是任意号码，它只是作为一个标志使用）；我们不像 FXS 那样使用注册方式向 FreeSWITCH 注册，而是使用“中继对接”方式与 FreeSWITCH 对接，因此我们不选注册复选框；接入方式选择“绑定”，并输入绑定号码（该绑定号码一般应该是运营商提供给我们的电话号码，但同样，也可以是做任意值，我们这里以 88888888 为例）。

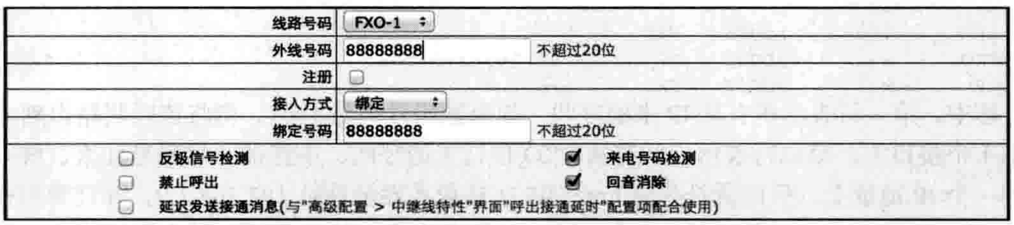


图 14-7　MX8 网关 FXO 接口配置

通过使用绑定方式，我们可以在 FXO 接口有来话时自动路由到 FreeSWITCH 上（后面

⊖ 通常的解决办法是去掉该行规则，如果拨打真正的外网的 120 时拨 0120 并通过相关的号码变换规则实现。

会讲到具体配置)。另外还可选择二次拨号的方式，我们先来讲一下二次拨号的方式。

注意，FXO 接口的功能相当于一个电话机，如果有人拨打该外线对应的电话号码（为了简单起见，我们以 88888888 为例），它是无法知道我们在该外线上到底接了一个电话机还是模拟网关的。当然，如果我们直接在外线接了一个电话机，有人打电话时就会振铃，我们就可以接听了。但是这里我们接了一个模拟网关，如果有电话呼进来，模拟网关振铃的铃流电压后就可以代替我们接听。但接听后下一步该怎么办呢？一般来讲，它可以给用户再放一个拨号音（有的网关可以在这里设置一个比较简短的语音提示，称为 IVR），这样主叫用户就可以再次通过 DTMF 按键进行拨号以拨打我们的内线号码。

当然，既然我们已经有了 FreeSWITCH，就可以用 FreeSWITCH 做任何我们想做的事。所以这里我们选择绑定方式。在绑定号码 88888888 后，对于任何外线来话，该网关都会向 FreeSWITCH 发送 INVITE 消息（在设置了正确的路由的情况下），被叫号码为 88888888。FreeSWITCH 在收到后，就可以进入 Dialplan，播放 IVR，引导来电用户进行下一步的操作。在这种情况下，FreeSWITCH 需要先对来电应答（发送 SIP 200 OK 消息），这样，网关收到 200 OK 消息后才应答进来的呼叫（相当于我们摘机），并建立通话。

当然，FreeSWITCH 也可以自己直接将该来电桥接（bridge）到一个内线号码号码上，让该内线号码振铃。如果有多条外线，可以分别插在不同的 FXO 接口上，对内也可以一一对应不同的内线电话（分机）。当外面的人拨打这些外线号码时，电话就能从某一个 FXO 接口进来，并最终向对应的内线电话振铃。这种方式，就称为 DID（Direct Inbound Dialing，对内直接呼叫）。同时这些外线号码就称为 DID 号码，简称 DID。广义上讲，即使对这些外线来话不是一一对应到内线分机上的，也称为 DID，在这种情况下，可以认为 DID 就是一个外线接入号，是一个被叫号码。

这里配置好了以后，我们还不足以建立它与 FreeSWITCH 通话。要灵活地对来话、去话进行处理，我们还要用到路由的概念。

4. 配置路由

MX8 网关通过路由机制对各种设备的来、去话进行控制。可以通过编辑路由表配置这些路由。路由表的语法也很简单。首先进入“拨号及路由”页面并添加如下的路由：

```
IP      1007    ROUTE   FXS     1
FXO     x       ROUTE   IP      192.168.1.9:5080
IP      x       ROUTE   FXO     5,6,7,8
```

其中，第一行表示所有从 IP 来的呼叫，如果被叫号码是 1007，则将该呼叫路由到 FXS 的第 1 个接口上；第二行表示，所有从 FXO 接口来的呼叫，不管被叫号码是什么，统一路由到一个 IP 地址上，后面就是我们 FreeSWITCH 服务器的地址 192.168.1.9，并且我们在这里使用 5080 端口，以避免对网关发来的 INVITE 请求进行鉴权（参考 9.2.2 节）。

下面，我们分几种情况进行说明。

1）模拟话机做主叫。当它发起呼叫时，由于我们在这里使用了向 FreeSWITCH 注册的方式，因而相当于网关设备把它转换成了一个 SIP 话机，网关设备就会向 FreeSWITCH 发

起 INVITE 请求，接下来的接续流程跟 SIP 话机做主叫是一样的，呼叫进入 FreeSWITCH 的 Dialplan，然后进行下一步的路由。

2）模拟话机做被叫。如果在 FreeSWITCH 中有人呼叫该模拟话机的号码（如 1007），FreeSWITCH 只是认为它就是一个普通的 SIP 用户，因而会找到网关设备注册时的 Contact 地址（即网关的 IP），并向该网关发 INVITE 请求。网关收到 INVITE 请求后，查找路由表，并匹配到上述路由表中的第一行，因而连接在相关的 FXS 口的模拟话机就会振铃。

3）接受模拟外线呼入。如果外面的电话从外线呼入，来话就到达网关的 FXO 接口。网关从上述路由表中查到第 2 行所示的路由，并向 192.168.1.9:5080 发送 INVITE 请求。FreeSWITCH 在收到 INVITE 请求后，会进入路由阶段，我们在日志中就可以看到类似如下的输出（还记得我们在 6.1.1 节讲到过的“绿色的行”吗？）：

```
Processing 139xxxxxxxx <139xxxxxxxx>->88888888 in context public
```

注意，其中的 88888888 就是我们设置的 DID。因此，为了能处理该通话，我们需要在 public Dialplan 中添加类似如下的配置：

```
<extension name="DID">
    <condition field="destination_number" expression="^88888888$">
        <action application="info" data=""/>
        <action application="ivr" data="welcome"/>
    </condition>
</extension>
```

上述配置可以在日志中显示来话 Channel 的相关信息，并执行 IVR，提示来电用户进行下一步操作。

4）通过模拟外线呼出。如果需要通过模拟外线呼出，则需要首先在 FreeSWITCH 中做一条路由，将去话路由到该网关上。在这种情况下，FreeSWITCH 将 MX8 网关看成一个外部网关，因而我们可以在 FreeSWITCH 中添加一个网关（其中 192.168.1.8 为 MX8 网关的 IP 地址）：

```
<gateway name="mx8">
    <param name="realm" value="192.168.1.8"/>
    <param name="register" value="false"/>
</gateway>
```

其中，FreeSWITCH 仅把 MX8 网关看作一个以中继方式对接的设备，因而不需要向其注册（register=false）。

然后，就可以使用如下的 Dialplan 将本地用户的去话路由出去。

```
<extension name="DID">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="bridge" data="sofia/gateway/mx8/$1"/>
    </condition>
</extension>
```

另外，在这种中继对接的模式下，我们也可以不在 FreeSWITCH 中添加网关，而是直接把去话送到网关的 IP 地址上，如：

```
<extension name="DID">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="bridge" data="sofia/external/$1@192.168.1.8"/>
    </condition>
</extension>
```

通过上面的 Dialplan 可以将 FreeSWITCH 中的电话送到 MX8 上，然后 MX8 收到请求后查找到以下的路由表项：

```
IP      x          ROUTE   FXO      5,6,7,8
```

该路由表表示，从 IP（以太网口）进来的呼叫，不管被叫号码是什么（x 代表任意号码），全部路由到 FXO 接口上，FXO 接口的选择顺序依次是 5、6、7、8。

注意

笔者使用的 MX8 有 4 个 FXS 口和 4 个 FXO 接口，FXO 接口的范围是 5 ~ 8。在呼出时，如果检测到某个接口忙，则顺序会选择下一个接口进行呼出。

5. 其他

对于网关来讲，还有其他的一些常用功能，比较典型的如主叫号码检测。对于一个从外线（FXO 接口）进来的来话而言，如果我们在外线上开通了来电显示功能，则可以在向 SIP 服务器发送 INVITE 之前检测主叫号码。

一般来说，现行的模拟线路都使用 FSK ⊖方式来传送主叫号码。主叫号码在第一声振铃和第二声振铃的时间隔内传送。所以，如果使用主叫号码检测，接续时间就会长一点。从图 14-7 中可以看出我们勾选了“来电号码检测”功能。

开启了来电号码检测后，如果外线没有开通来电显示功能（通常该功能作为运营商的增值业务，是要单独收费和开通的），则肯定检测不到主叫号码。MX8 网关在这时通常使用我们“绑定”的号码作为 SIP 侧的主叫号码。

另一个值得一提和功能是“回声消除”。回声一般是在话机等终端设备上产生的，典型的场景就是从话筒中传出的声音又传回到了麦克风里去了（相当于声音反射，称为回声），这样对端就会在电话中听到自己的声音，听起来会感觉不舒服⊜。如果设备有较好的回声消除功能，它可以检测麦克风的输入，如果里面包含话筒中刚刚输出过的内容（它应该能记住一段时间的内容），则自动在传到对方之间将这部分声音数据去掉。这种技术就称为回声消除。

在图 14-6 中我们也勾选了“回声消除”功能，它在检测到回声时能起到一定的作用。另外，一幅比较好的耳机也能大大减少回声（声音的输出直接传的耳朵里，很少会扩散到麦克风中）。

在本节我们以迅时 MX8 网关为例子讲解了模拟网关的配置以及与 FreeSWITCH 的对

⊖ Frenquency-shift keying，即移频键控。参见 http://en.wikipedia.org/wiki/Frequency-shift_keying。

⊜ 产生回声的原因是对端终端的问题，即如果你能听到回声，那一定是对端的终端设备将你的声音又反射了回来。

接。其他厂商的网关也有类似的功能，只是具体的配置和实现方式不同。具体的应用可以参考相应的设备手册。

14.4 通过 E1 线路与其他系统对接

前面，我们讲过通过 IMS 或模拟中继线连接 PSTN 网络的例子。但很多时候，我们还是需要使用 E1 与其他系统对接。很典型地，某些运营商可能只提供 E1 线路，或者某些设备只提供 E1 接口。

类似我们在 14.2.2 节讲到的拓扑结构和对接方式，我们也可以使用 E1 网关或板卡来与其他系统对接。在 14.2 节，我们只讲了使用网关方式对接的例子。在这里，我们把两种对接方式都讲一下，以帮助读者有一个全面的了解。

我们将按图 14-8 所示的拓扑结构来搭建实验环境。其中 FS1 和 FS2 分别是两台装有 FreeSWITCH 的主机，它们分别具有 IP 地址 192.168.1.115 和 192.168.1.119。FS1 上装有 E1 板卡，可以通过 E1 线[1]与 E1 网关相连，同时 E1 网关又通过以太网（使用 SIP 协议）与 FS2 相连。

其中 FreeSWITCH 我们使用默认的安装和默认的配置。下面我们就来看一下需要经过哪些步骤可以让分别注册到 FS1 和 FS2 上的 SIP 用户互相打电话。

14.4.1 配置 FS1

首先，我们来看一下 FS1 上的配置。FS1 上需要安装硬件板卡，安装驱动程序等，因而这一部分配置是比较复杂的。而且我们也将在这里学到一些新的名词术语和概念等。

1. 背景资料

传统的电话交换机都是以时分复用（TDM，Time Division Mutiplex）的方式工作的，因此，与传统交换机相连的设备及板卡都又统称为 TDM 设备或 TDM 板卡。在 FS1 上，我们需要使用 E1 板卡来支持 E1 接口。生产这类 E1 板卡的厂商有 Sangoma 和 Digium 等（当然，它们也生产模拟板卡），国内也有一些生产这种兼容卡的厂商，如 OpenVox 等。从历史上来讲，Digium 成立于 1999 年，算是老牌子的板卡厂商，也就是它推动了 Asterisk 开源 PBX 软件的发展。Sangoma 是一家创立于 1984 年的 VoIP 系统供应商，也是相当有历史的。Sangoma 从 FreeSWITCH 诞生起就一直与 FreeSWITCH 社区深度合作，并且贡献了 FreeTDM 软件和 mod_freetdm 模块，用于配合各种硬件板卡在 FreeSWITCH 上工作。

最初，这些硬件板卡都是靠 Zaptel[2]来驱动的，FreeSWITCH 中有一个对应的模块叫

---

[1] 设备之间对接需要使用 E1 交叉线。E1 线由 4 条线组成，其中 2 条是收（接收），2 条是发（发送）。交叉线的含义就是一端的“收”要接到对端的“发”上。物理接口一般有两种，一种是 RJ48（物理上跟 RJ45 相同，只是线序不同）双绞线接口，阻抗为 120Ω；另一种是同轴电缆接口，阻抗是 75Ω。也有在这两种物理接口转换的适配电缆。

[2] Zaptel 指 Jim Dixon 的开放电脑电话硬件驱动 API，后来 Digium 公司生产了相关的板卡并改进了该 API。参见 http://www.voip-info.org/wiki/view/Zaptel。

mod_openzap[1]。后来，Sangoma公司贡献了并维护着mod_freetdm模块，因此就很少有人用mod_openzap了。同时，Digium也将Zaptel更名为DAHDI[2]，并继续进行改进。该协议主要在Digium的板卡中使用，国内也有一些"Asterisk兼容卡"支持DAHDI协议。Sangoma公司的板卡除了支持Zaptel和DAHDI外，还支持自己的驱动Wanpipe。并且，一般认为在并发数较高的情况下Wanpipe协议比DAHDI表现要好。

FreeTDM作为Sangoma公司贡献的开源软件，可以单独使用，也可以与FreeSWITCH配合使用。而与FreeSWITCH的配合就是靠mod_freetdm将它们联系起来的。

在本质上，FreeTDM也是一个模块化的结构，我们可以在源代码目录中使用ls命令列出FreeTDM支持的子模块[3]。在FreeSWITCH源代码目录中执行ls命令的输出结果如下：

```
$ ls libs/freetdm/src/ftmod/
ftmod_analog/     ftmod_isdn/      ftmod_pika/      ftmod_sangoma_isdn/   ftmod_wanpipe/
ftmod_analog_em/  ftmod_libpri/    ftmod_pritap/    ftmod_sangoma_ss7/    ftmod_zt/
ftmod_gsm/        ftmod_misdn/     ftmod_r2/        ftmod_skel/
```

上面列出的这些子模块基本上可以分为两种——信令模块和IO模块。其中前者主要负责通话的建立和释放（如ftmod_isdn)，类似于我们已经熟悉的SIP；而后者主要负责话音数据的输入输出（如ftmod_wanpipe)，类似于我们熟悉的RTP。

在IO层，FreeTDM可以有两种工作模式，一种是DAHDI模式，它是由ftmod_zt驱动的，主要兼容DAHDI及Zaptel协议的板卡；另一种为Wanpipe模式，主要用于Sangoma生产的板卡。这里我们以Sangoma板卡为例来进行说明，因此，主要讲解Wanpipe模式。

关于Sangoma板卡的安装，涉及编译内核、安装各种驱动程序，以及进行各种配置等，操作起来比较复杂。下面的内容假定已经安装好了板卡及驱动。

2. 安装和配置 mod_freetdm

只有正确安装了Wanpipe及ISDN信令协议后才能在编译mod_freetdm时正确编译相关的模块。因此，需要确保上述步骤成功才能执行这一步。

跟安装其他模块类似，在FreeSWITCH源码目录直接执行以下命令就可以完成mod_freetdm的安装：

```
# make mod_freetdm
# make mod_freetdm-install
```

安装过程中安装脚本可以自动找到相应的驱动程序，并配置安装相关的FreeTDM子模块。

安装完成之后，可以在FreeSWITCH的conf目录中找到两个与FreeTDM相关的文件。其中，conf/freetdm.conf为FreeTDM的配置文件，内容如下：

```
[span wanpipe wp1]
```

---

[1] 注意，该模块的源代码不在src/mod目录下，而是在libs/openzap目录下。

[2] DAHDI即Digium Asterisk Hardware Device Interface的缩写。由于注册商标问题，Zaptel于2008年更名为DAHDI。参见http://blogs.digium.com/2008/05/19/zaptel-project-being-renamed-to-dahdi/。

[3] 为了与FreeSWITCH中的模块相区别，我们在这里称FreeTDM的模块为"子模块"。

```
trunk_type => e1
group=1
b-channel => 1:1-15
b-channel => 1:17-31
d-channel => 1:16
```

该配置文件的格式是类似于 Windows 中“.ini”文件的配置格式。其中第 1 行配置了一个 Span，一个 Span 相当于板卡上的一个 E1 端口，它的类型是 wanpipe，名称是 wp1；trunk_type 即中继的类型，这里是 e1；group 表示一个组号；该 E1 板卡上共有 32 个时隙（又称信道，即 Channel），其中 0 时隙是传时钟同步的，在这里没有列出，16 时隙为 D 信道（传信令），其他的 30 个时隙为 B 信道（传话音）。

另一个配置文件是 mod_freetdm 的模块配置文件，它位于 conf/autoload_configs/freetdm.conf.xml。具体内容我们就不详细列举了，其中一部分如下：

```
<sangoma_pri_spans>
  <span name="wp1" cfgprofile="my_pri_nt_1">
    <param name="dialplan" value="XML"/>
    <param name="context" value="default"/>
    </span>
</sangoma_pri_spans>
```

其中，它定义了一个 Span，名称 wp1 是与 freetdm.conf 中的 Span 的名字相对应的。并且，里面还定义了 Dialplan 和 Context，用于对该 E1 中继的来话进行路由。

当然，读者可以发现这些配置的值都是我们在安装驱动时的配置向导生成的。后面也可以手工更改并重新加载。

确保 wanrouter 已经启动，然后就可以在 FreeSWITCH 控制台上使用如下命令加载 mod_freetdm 了：

```
freeswitch> load mod_freetdm
+OK Reloading XML
+OK
```

至此，安装的工作已经完成，并且看起来 mod_freetdm 也加载成功了。

加载 mod_freetdm 之后，可以在 FreeSWITCH 控制台上使用模块提供的 ftdm 命令查看 E1 板卡及各 Span 和 Channel 的状态，比如，使用以下命令列出当前系统配置的所有 Span：

```
freeswitch> ftdm list

+OK
span: 1 (wp1)
type: Sangoma (ISDN)
physical_status: alarmed
signaling_status: DOWN
chan_count: 31
dialplan: XML
context: default
dial_regex:
fail_dial_regex:
hold_music:
analog_options: none
```

从上面的输出可以看到，我们只有一个Span（wp1），它的物理状态（physical_status）是alarmed（警告），信令状态（signaling_status）是DOWN，这意味着未连接。

如果这时候插入自环电缆，则可以看到FreeSWITCH控制台上会输出好多DEBUG级别的日志信息，标志状态的变化。重复上述命令会看到物理状态变成ok，但是信令状态仍然是DOWN(因为自环只能模拟一个“对端设备”，可以模拟物理连接的情况，无法与“对端”建立信令连接）。插入自环电缆后查看状态的输出如下：

```
freeswitch@ubuntu32> ftdm list

+OK
span: 1 (wp1)
type: Sangoma (ISDN)
physical_status: ok
signaling_status: DOWN
chan_count: 31
dialplan: XML
context: default
dial_regex:
fail_dial_regex:
hold_music:
analog_options: none
```

另外，也可以使用“ftdm dump”命令显示实际的Span中指定的时际的状态信息，如：

```
freeswitch> ftdm dump 1 1
+OK
span_id: 1
chan_id: 1
physical_span_id: 1
physical_chan_id: 1
physical_status: ok
physical_status_red: 0
physical_status_yellow: 0
physical_status_rai: 0
physical_status_blue: 0
physical_status_ais: 0
physical_status_general: 0
signaling_status: DOWN
type: B
state: DOWN
last_state: DOWN
txgain: 0.00
rxgain: 0.00
cid_date:
cid_name:
cid_num:
ani:
aniII:
dnis:
rdnis:
cause: NONE
session: (none)
```

“ftdm dump”命令可以将当前 Span 和 Channel 的状态显示输出，上面的例子中后面两个参数分别是 Span 号和 Channel 号。当然，如果省略 Channel 号，则会 dump 整个 Span 里面所有的 Channel。

3. 配置电话路由

虽然我们还没有与其他系统对接，但这里我们先把来、去话的路由准备好，等对端的设备准备好以后就可以直接打电话进行测试了。

注意，这里说的“来、去”的概念都是相对的——一路通话，对于一端来说是来话，对于另一端来说就是去话。我们这里涉及的设备比较多，因而一定要想清楚来话和去话是针对哪个设备来说的。

在 FS1 上，对于从 E1 上的来话而言，由于我们上面已经指定了将来话路由到 XML Dialplan 的 default Context 中，因此暂时不需要任何设置。

对于去话，我们在 default Context 中配置以下 Dialplan，让本地用户在拨打以 0 开头的电话号码时将电话从 E1 端口上送出去。Dialplan 的内容如下：

```
<extension name="E1">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="bridge" data="freetdm/1/a/$1"/>
    </condition>
</extension>
```

其中，正则表达式“0(.*)”表示匹配所有以被叫号码 0 开头的通话。如果匹配，则将实际的电话号码存储到“$1”变量中（不包括 0）。然后，执行后续的 Action。这里的 Action 只有一个 bridge。其中，bridge 是一个 App，对此我们大家都很熟悉了。后面的 freetdm 为 mod_freetdm 提供的呼叫字符串，各参数是以“/”分隔的，其中 freetdm 为 Endpoint 的类型，后面的参数含义如下：

- “1”代表第一个 Span ⊖。我们在这里只有一个 Span。
- “a”代表选线规则。其中，小写的“a”表示从小到大选择一个空闲的时隙（Channel）并将电话送出，大写的“A”表示从大到小选择一个空闲时隙。注意如果双方同时选择一个时隙，将会发生“撞车”的现象。为了最大限度避免这种碰撞，相互对接的双方可以从不同的方向优先选择时隙。当然，如果真发生了“撞车”，也有相关机制保证有一方会自动放弃并选择下一个可用的时隙。
- “$1”表示实际送出的被叫电话号码。其中，从 Dialplan 的正则表达式中可以看到，“$1”的值中是不包括“0”的，即如果 FS1 上的用户拨叫了 01000，实际送到 E1 线路上的被叫号码将是 1000。这是一种常见的做法，通俗的说法就是在电话路由的时候“吃”掉前面的第一个“0”。

至此，FS1 上的配置就结束了。我们再来看其他设备的配置。

⊖ 在主机上装有多个 Span 的情况下，也可以将任意的时隙在逻辑上组成一个分组，因此这里也可以填入一个组名。一个分组可以跨越多个 Span，在这里就不多介绍了，有兴趣的读者可以参见 http://wiki.freeswitch.org/wiki/FreeTDM。

14.4.2 配置 E1 网关设备

在这里，我们以鼎信通达的 E1 网关 MTG1000 为例。MTG1000 是一款 E1 转 SIP 的网关设备（它只是完成 E1/SIP 话路的信令转换和路由，因此，我们还需要另外一个 SIP 服务器 FS2 才能完成通话测试）。

在进行下一步的测试之前，先使用 E1 交叉线将 MTG1000 和 Sangoma A101 连接起来。

该设备默认有两个以太网口。其中，网口 0 的默认地址是 192.168.1.111，用于与其他 SIP 设备对接；网口 1 的默认地址是 11.11.11.11，一般用于连接笔记本电脑以便对设备进行管理。

1. 添加 PRI 中继

打开浏览器，进入管理界面，选择添加一个 PRI 中继，如图 14-8 所示。



图 14-8 添加 PRI 中继

其中，中继编号使用默认的“0”；中继名称可以为“FS-ISDN”；接口标志符使用“0”；D 通道选择“启用”；E1/T1 端口号使用默认的“0”；协议类型选择“ ISDN”（与我们 FS1 中的 EuroISDN 对应）；接口属性选择“设备侧”（与 FS1 中的相对，对方应该选择 NET，即网络侧）；其他选项可以使用默认值。

单击“确定”按钮后保存设置。然后可以转到状态页面查看 E1 中继端口的状态。如图 14-9 所示，如果看到端口状态变成绿色的（状态是 Actived）则表示链路正常了。

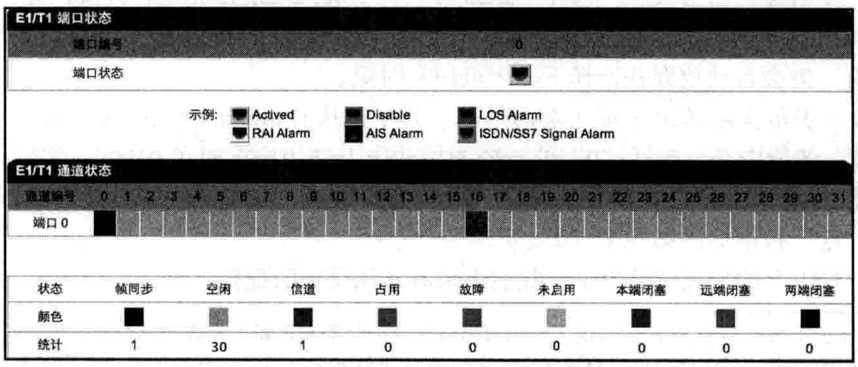


图 14-9 E1 中继状态

这时候也可以在 FS1 中使用 ftdm list 和 ftdm dump 等命令检测链路的状态。

2. 添加 SIP 中继

与 PRI 中继相对应，在 MTG1000 中，与我们的 SIP 服务器 FS2 相连的 SIP 链路称为 SIP 中继。

如图 14-10 所示。在管理界面上选择添加 SIP 中继，中继编码使用默认的 0；中继名称为 FS-SIP；对端地址输入我们 FS2 的 IP 地址，在这里是 192.168.1.119；对端端口使用 5080，以防止对该网关设备进行鉴权，简化配置。

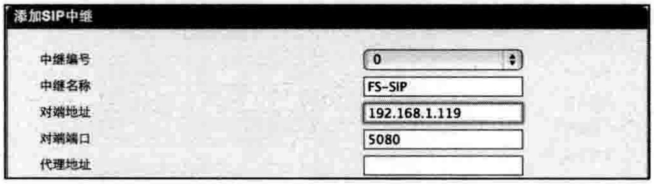


图 14-10　添加 SIP 中继

其他的配置选项都使用默认的配置即可。单击“确定”按钮保存配置。

3. 添加路由

有了 E1 中继和 SIP 中继，该网关的作用就是把这些中继串起来。而这种“串”就是靠 MTG1000 的由路功能实现的。

首先，我们添加一条路由，将从 E1 端口进来的通话路由到 SIP 中继上去。在 MTG1000 的管理界面上选择添加 PSTN-IP 路由，如图 14-11 所示。

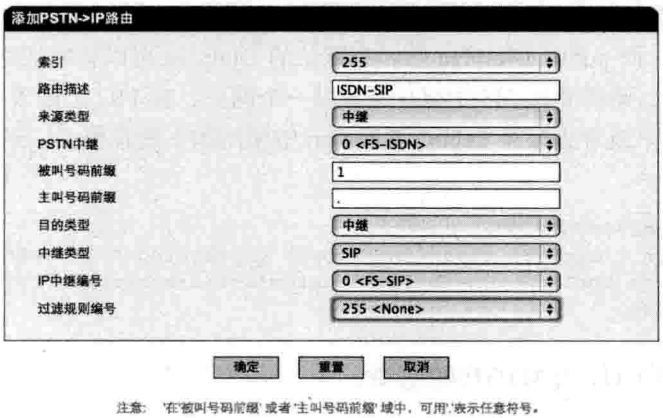


图 14-11　添加 PSTN 到 SIP 路由

其中，索引号使用默认的 255；路由描述中我们填上“ISDN-IP”；来源类型选择“中继”；PSTN 中继选择我们刚刚创建的“0”；被叫号码前缀填入“1”，即从该 E1 线路进来的，而且被叫号码为 1 的电话将走这一条路由；主叫号码前缀由于我们不关心，这里可以填入“.”

以接受任意主叫号码；目的类型选择“中继”；中继类型选择“SIP”；IP 中继编号即我们刚刚创建的 SIP 中继“0”；其他参数使用默认值。

上面的配置的目的就是，从 E1 来的呼叫若满足一定的主、被叫号码规则，则路由到一条 SIP 中继上去。

用类似的方法可以配置一条 SIP 到 E1 的中继，让从 SIP 侧过来的通话能送到 E1 上去。如图 14-12 所示，参数的含义与上面的类似，在这里就不多介绍了。

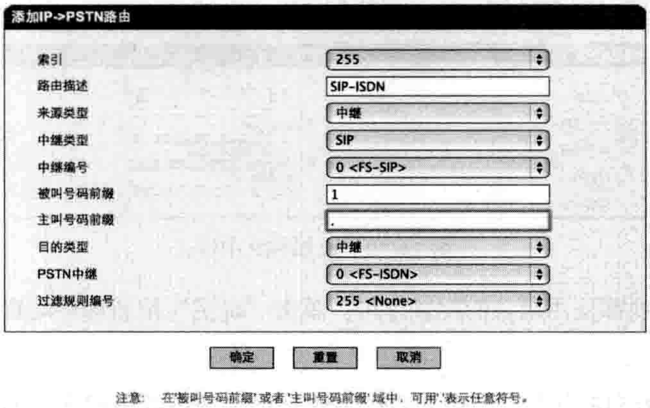


图 14-12 添加 SIP 到 PSTN 路由

14.4.3 配置 FS2

在 FS2 侧，对于来话而言，由于 MTG1000 将电话路由到了 5080 端口，因此来话将进入 public Dialplan，而 public Dialplan 已经有默认的 Dialplan 可以将电话路由到本地用户了。

对于去话而言，则需要把 MTG1000 也看成一个网关，在 FS2 上配置一个网关并将电话路由到该网关即可。或者直接在 default Dialplan 中使用如下配置将去话送到 MTG1000 所在的 IP 地址上：

```
<extension name="MTG1000">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="bridge" data="sofia/external/$1@192.168.1.111"/>
    </condition>
</extension>
```

其中，192.168.1.111 为 MTG1000 的 IP。

1. 拨打测试

我们上面配置了 FS1、MTG1000 网关及 FS2，下面可以进行拨打测试了。如果 FS1 上的用户 1000 想呼叫 FS2 上的用户 1001，则拨打 0 加上对方的号码，如 01001；电话在 FS1 上进行路由时被叫号码最前面的 0 会被 Dialplan“吃”掉，并将剩下的 1001 送到 E1 中继网关 MTG1000 上；MTG1000 在收到来自 E1 中继的来话后，查找本地的路由表，并将该电话

送到配置好的 SIP 中继上，实际上就是送到我们的 FS2 的 5080 端口上；FS2 在 5080 端口上接收到来话后，即查找本 public Dialplan，将最终向本地用户 1000 振铃，1000 接起电话后就可以进行通话了。呼叫流程如图 14-13 所示。

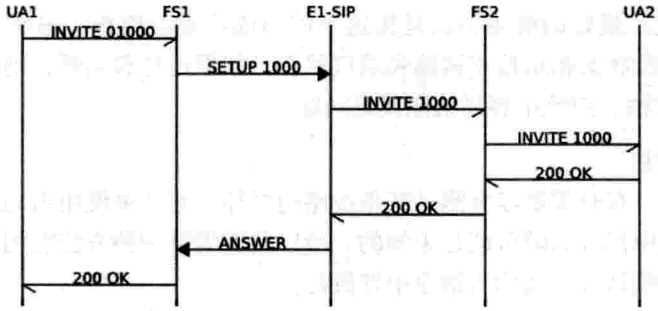


图 14-13 呼叫流程

反过来从 FS2 到 FS1 的通话流程跟上面差不多，读者可以自行推演。

通话过程中，读者也可以使用“ftdm dump”命令查看当前 Span 的占用状态。如：

```
freeswitch> ftdm dump 1 5

span_id: 1
chan_id: 5
physical_span_id: 1
physical_chan_id: 5
physical_status: ok
physical_status_red: 0
physical_status_yellow: 0
physical_status_rai: 0
physical_status_blue: 0
physical_status_ais: 0
physical_status_general: 0
signaling_status: UP
type: B
state: UP
last_state: PROCEED
txgain: 0.00
rxgain: 0.00
cid_date:
cid_name:
cid_num: 1000
ani: 1000
aniII:
dnis: 1001
rdnis:
cause: NONE
session: c854e884-1c97-11e3-befe-0f1ace9d0d7b

-- States --          -- Function --             -- Location --          -- Time Offset --
DOWN   => RING     [sngisdn_process_con_ind]   [ftmod_sangoma_isdn_stack_hndl] 0ms
```

```
RING  => PROCEED [channel_on_routing]        [mod_freetdm.c:456]          5ms
PROCEED => UP    [channel_receive_message_b] [mod_freetdm.c:991]         10ms

Time since last state change: 91886ms
```

如果在测试的过程中，出现电话不通的情况，那么很可能是配置的时候漏掉了某些地方或写错了某些参数，最好的解决办法是按这里的描述再重新检查一遍所有的配置，根据界面上（见图 14-9）的状态指示检查链路和端口状态。如果还是有问题，则复习一下我们在第 10 章里讲的排错方法，应该很容易就能找到问题。

2. 指定中继拨打

在实际应用中，有时需要经常测试某条线路的好坏。而如果使用出局路由自动选线的方式选择到的某一条中继或时隙可能是未知的。所以我们需要一种方法能测试指定选择的某条中继或时隙，这种测试方法就称为指定中继拨打。

我们先来看如下的 freetdm 呼叫字符串，在这里不使用“a”或“A”的选线方式，而是直接指定从某个时隙呼出，如：

```
freeswitch> originate freetdm/1/5/1000 &echo

+OK c854e884-1c97-11e3-befe-0f1ace9d0d7b
```

上述命令从第一个 Span 的第 5 个时隙呼出，并在本端执行 echo，对端接听后可以听到自己的回声。

如果我们不挂机再重复执行一次，可以看到由于该时隙已被占用，并返回拥塞消息 NORMAL_CIRCUIT_CONGESTION：

```
freeswitch> originate freetdm/1/5/1000 &echo

-ERR NORMAL_CIRCUIT_CONGESTION
```

了解了上述方法以后，我们可以设置以下 Dialplan 进行指定中继拨打。

```
<extension name="Dial specific trunk">
    <condition field="destination_number" expression="^\*22([0-9])([0-9])(.*)$">
        <action application="bridge" data="freetdm/$1/$2/$3"/>
    </condition>
</extension>
```

其中，正则表达式为“^*22([0-9])([0-9])(.*)$”。如果我们呼叫“*22151000”，则它与正则表达式匹配。其中，“*22”相当于我们指定中继拨打的功能码；接下来的 1 匹配第一个括号里的“[0-9]”，被记到变量“$1”中；同理，“5”被记到变量“$2”中；后面的“1000”被记到变量“$3”中。因此，最后呼叫字符串“freetdm/$1/$2/$3”将变成“freetdm/1/5/1000”，即从第一个 Span 的第 5 个时隙呼出。

如果我们想从第 6 个时隙呼出，则可以呼叫“*22161000”。通过这种方法，可以呼叫任意的时隙[⊖]，这便是指定中继拨打的含义。

⊖ 实际上，这里的 Dialplan 不支持呼叫 2 位数的时隙，读者可以自己练习一下，看能否通过所学的知识自己实现。

14.4.4　对接其他厂家的 E1 网关

作为一个对比，我们也来看一下与迅时 E1 数字网关的对接。这里我们使用的设备型号是 MX100-TG。其中 FS1 与 FS2 的配置保持不变。在 MX100-TG 上，我们也需要与 14.4.2 节描述的一样——设置 ISDN 和 SIP 侧的参数、添加双向的路由等。不过，后面我们可以看到，迅时网关的配置步骤比较简洁。

1. ISDN 设置

MX100-TG 的配置界面与 MX8 类似。打开浏览器，进入管理界面，选择 ISDN，便看到如图 14-14 所示的配置界面。



图 14-14　添加 ISDN 中继

其中，ISDN 链路的名称我们可以为 FS-ISDN；收号方式选择默认的整体收号；D 通道选择默认的 16 时隙；接口协议选择默认的用户侧（与 FS1 中的相对，对方应该选择 NET，即网络侧）；信令标准选择默认的 CCITT；选线方式可以选择默认的顺序，也可以选择其他的选线方式；其他选项可以使用默认值。

单击“提交”按钮后保存设置，设置即可生效。如果一切正常，则 E1 链路就会变成正常的连通状态，在图 14-14 所示的界面上也能观察到。

2. 路由设置

迅时系列的网关，可以直接将 SIP 通话路由到 IP，因而不需要设置明确的 SIP 中继。为了能让 FS1 与 FS2 之间互打，我们只需简单地设置以下两条路由：

```
ISDN      x      ROUTE      IP      192.168.1.119:5080
IP        x      ROUTE      ISDN    1
```

其中，第一条与在模拟网关 MX8 中的设置类似，即所有从 ISDN 侧来的呼叫，不管被叫号码是什么，全部（通过 SIP）路由到 FS2 的 IP 地址上去（192.168.1.119 : 5080）；而第二条也很容易理解，即所有从 IP 侧来的 SIP 呼叫，全部路由到第 1 条 ISDN 链路（E1）上去（如果有多个 E1 的话，也可以使用类似模拟线的配置方式，如“1,2,3,4”选择不同的 E1 链路）。

14.5 对接 Asterisk

Asterisk 是老牌的开源 VoIP 软件。事实上，FreeSWITCH 的作者 Anthony Minessale 在早些年也是 Asterisk 的开发者，后来由于对 Asterisk 的架构设计和性能问题有意见，提议开发了一个新的 Asterisk 分支，但是开发新分支的想法却未得到 Asterisk 社区主要负责人的支持，因此便从头开发了 FreeSWITCH ⊖。

由于 Asterisk 比 FreeSWITCH 资格老，因此在国内有大量的用户群，在很多情况下都需要与 FreeSWITCH 进行对接。因此，这里我们也来看一下 Asterisk 与 FreeSWITCH 对接的例子。

在本例中，我们仅使用中继方式（在 Asterisk 中称为 peer，即对等的连接方式）与 Asterisk 进行对接，其他的对接方式如相互向对方注册等留给读者练习。我们假定读者已经熟悉 Asterisk。先在 Asterisk 上添加一个分机 6000，并用一个 SIP 客户端注册上去，为下面的测试做准备。另外，下面的测试中 FreeSWITCH 的 IP 地址为 192.168.7.5，而 Asterisk 的 IP 地址为 192.168.7.99。

14.5.1 从 FreeSWITCH 呼叫 Asterisk

首先，我们已经很熟悉如何在 FreeSWITCH 中将电话送出，通过使用下列命令可以呼叫 Asterisk 上的 6000：

```
freeswitch> originate sofia/external/6000@192.168.7.99 &echo
```

或者，使用如下的 Dialplan，当 FreeSWITCH 上的用户拨打 6000 时，也可以送到 Asterisk 上：

```
<action application="bridge" data="sofia/external/6000@192.168.7.99"/>
```

呼叫到达 Asterisk 上后，就可以在 Asterisk 的控制台上看到类似如下的日志：

```
NOTICE[3684][C-0000000d]: chan_sip.c:25381 handle_request_invite:
```

⊖ 在此，笔者只是陈述一个事实，并无意将两者对立起来。其实，Asterisk 的新版本在近几年性能也有了很大的提升。并且，在市场上有很多成熟的应用，很多用户也将会长期使用 Asterisk，这也是笔者写这一节的原因之一。Anthony Minessale 在 2010 年接受 VOIPToday 采访并被问及 FreeSWITCH 与其他“竞争者”的关系时也说过：这并不是一个 FreeSWITCH 必须“打败”Asterisk 才能“胜利”的公认的所谓的“零和游戏”，在很多情况下人们都会把 FreeSWITCH 用作他们已有的 Asterisk、CallWeaver、YATE 安装部署的一部分……

```
Call from '' (192.168.7.5:5080) to extension '6000' rejected
    because extension not found in context 'public'.
```

出现上述提示的原因是我们没有在 Asterisk 上配置入局路由，因而我们的呼叫被拒绝（rejected）了。接下来，我们只需打开 Asterisk 的配置文件 extension.conf，在 public Context 中增加如下的设置就可以了。

```
exten => 6000,1,Dial(SIP/6000)
```

其中，“6000”匹配被叫号码；“1”是一个优先级标志；“Dial”是一个 Application，类似 FreeSWITCH 中的 bridge；最后“SIP/6000”是一个呼叫字符串。

在 Asterisk 的控制台上使用 dialplan reload 命令使用之生效，然后，再一次在 FreeSWITCH 中发起呼叫，Asterisk 上的分机 6000 就可以振铃了。

14.5.2　从 Asterisk 上呼叫 FreeSWITCH

我们也可以很容易在 Asterisk 中使用如下 Dialplan 将电话送到 FreeSWITCH 上。

exten => _01X.,1,Dial(SIP/${EXTEN:1}@192.168.7.5:5080)

其中，“_01X.”是 Asterisk 中的模式匹配规则，它匹配以“01”开头的被叫号码；“1”为优先级标志；而后面的 Dial 相当于 FreeSWITCH 中的 bridge App；再后面是一个以 SIP 开头的呼叫字符串；“${EXTEN}”即模式匹配中的被叫号码，而“${EXTEN:1}”即去掉最前面的一位（即“吃掉 0”），然后把剩下的号码作为被叫号码并将呼叫送到 FreeSWITCH 的地址 192.168.7.5:5080 上去。

因此，使用上述 Dialplan 在 Asterisk 上使用分机 6000 呼叫号码 01000 时，Asterisk 就会“吃”掉被叫号码中最前面的“0”，并将呼叫送达 FreeSWITCH，然后 FreeSWITCH 上的分机 1000 就会振铃。

上述呼叫在 FreeSWITCH 中能够正确路由是因为 FreeSWITCH 中默认的 publicDialplan 包含了到用户 1000 的路由。当然，如果是呼叫其他号码，则也可以根据实际情况在 FreeSWITCH 中配置相关的入局路由。

14.5.3　其他

在上面的例子中，我们仅介绍了使用中继对接的最简单的情形。电话是可以呼通了，但出于用于生产环境中的安全性考虑，就需使用账号或 IP 地址对来话进行验证了，这部分内容就留给读者自行练习了。当然，读者也可以参考以下 Wiki 页面上的对接说明 https://wiki.freeswitch.org/wiki/Connecting_Freeswitch_and_Asterisk。

另外，从上面的例子可以看出，在 Asterisk 与 FreeSWITCH 中的用户及路由的配置概念都是很类似的。如果用户熟悉 Asterisk，则 FreeSWITCH 也应该很快能上手。Wiki 页面 http://wiki.freeswitch.org/wiki/Rosetta_Stone 对两者相似的概念进行了对比，对于刚刚从 Asterisk 转到 FreeSWITCH 的读者来说，读一读或有帮助。

最后，关于 Asterisk 和 FreeSWITCH 的关系，我们来看一下 Anthony Minessale 在 2009

年接受媒体采访时的一段对话[1]：

媒体：如果有人想把他或她的 Asterisk 系统换成 FreeSWITCH，是否需要编写很多的代码以及学习完全不同的配置文件语法？

Anthony：并不像你想象的那么难。理解 FreeSWITCH 最关键的地方有一个它与 Asterisk 之间的词形变化表。事实上，它们之间有很多相似的地方，如它们都使用可加载模块、都使用“application”执行相应的功能等。不同的，同时也是对学习 FreeSWITCH、至关重要的一点是，其尝试最大程度地去除一切复杂性，我们可从一个简单的角度看待 Free SWITCH。很多从 Asterisk 转移过来的 FreeSWITCH 新手往往由于想得太多而把问题搞得很复杂。实际上我们无比强大的社区成员已经写了一个名为“Asterisk 罗塞塔石碑[2]”的 Wiki 页面[3]，该页面非常好地解释了从 Asterisk 转到 FreeSWITCH 的各种概念和要素。另外，我们也有一个模块[4]允许你使用你熟悉的 Asterisk 的语法写 Dialplan 进行电话路由。

14.6 使用 H.323 协议对接

H.323 是比较古老的 VoIP 协议，然而现在还是有很多设备支持该协议。作为一款多协议的 VoIP 系统，FreeSWITCH 也支持 H.323。下面我们就来简单讨论一下如何在 FreeSWITCH 中使用 H.323。

FreeSWITCH 中有两个 H.323 的实现——mod_opal [5]和 mod_h323 [6]。前者基于 H323plus 库[7]，后者基于 Opal [8]库，两个库又都依赖于 ptlib [9]。

14.6.1 mod_h323

我们先来看 mod_h323。下面我们将讨论如何对 mod_h323 进行安装、配置和拨打测试。

1. 安装

在安装 mod_h323 之前，首先要安装它依赖的库。这里需要注意的问题是，mod_h323 依赖的两个库 ptlib 和 h323plus 需要特定的版本配合在一起才能正常工作。该模块对应的

[1] 来自 http://www.888voip.com/interview-with-anthony-minessale-from-freeswitch/。

[2] Rosetta Stone，是一块刻有古埃及法老托勒密五世（Ptolemy V）诏书的石碑。由于这块石碑同时刻有同一段内容的三种不同语言版本，使得近代的考古学家得以有机会对照各语言版本的内容，解读出已经失传千余年的埃及象形文字的意义与结构。参见：http://zh.wikipedia.org/wiki/罗塞塔石碑。——笔者（译者）注

[3] 参见 http://wiki.freeswitch.org/wiki/Rosetta_stone 。

[4] 指 mod_dialplan_asterisk。——笔者（译者）注。

[5] 参考 http://wiki.freeswitch.org/wiki/Mod_opal。

[6] 参考 http://wiki.freeswitch.org/wiki/Mod_h323。

[7] 参考 http://www.h323plus.org/。

[8] 参考 http://www.opalvoip.org/。

[9] ptlib 是一个底层的支持库，提供一些抽象和跨平台的支持。真正的历史是这样的：所有这些都起源于一个开源的项目 OpenH323，后来，分离成了两个项目 Opal 和 H323plus，底层的一些抽象及跨平台支持也几乎在同时成了独立的库，叫 PWlib，后改名为 ptlib。

Wiki 页面上有几个特定的组合列表。这里笔者使用的是 h323plus 官方网站上最新的组合。

笔者是在 Ubuntu Linux 12.04 上进行测试的，在编译上述两个库之前，要确保系统上有 flex 和 bison 两个软件。可以用如下命令安装它们：

```
# apt-get install flex bison
```

然后，使用如下命令下载 ptlib 和 h323plus：

```
wget http://www.h323plus.org/source/download/ptlib-2.10.9.tar.bz2
wget http://www.h323plus.org/source/download/h323plus-v1_25_0.tar.gz
```

解压并配置安装 ptlib：

```
# tar xvjf ptlib-2.10.9.tar.bz2
# cd ptlib-2.10.9
# ./configure && make && make install
```

解压并配置安装 h323plus：

```
# tar xvzf h323plus-v1_25_0.tar.gz
# cd h323plus
# ./configure && make
```

在 make 过程中，笔者遇到以下错误：

```
/home/app/book/h323/h323plus/include/openh323buildopts.h:37:34:
fatal error: ptlib/../../revision.h: No such file or directory
```

为了解决该问题，笔者修改了对应的文件（h323plus/include/openh323buildopts.h"），将第 37 行的“#include <ptlib/../../revision.h>”一行注释掉（或删除）。然后，继续配置安装：

```
# make && make install
```

上述两个库安装完毕后，到 FreeSWITCH 的源代码目录下安装 mod_h323。由于我们上面的步骤中两个库的安装位置都在 /usr/local 目录下，因此我们需要改变 mod_h323 的 Makefile（src/mod/endpoint/mod_h323/Makefile），将里面所在的“/usr”路径都替换为“/usr/local”。修改完毕后，就可以在 FreeSWITCH 源代码目录中使用如下命令安装了：

```
# make mod_h323-install
```

或者，进入 mod_h323 源代码目录上，使用如下命令安装：

```
# cd src/mod/endpoint/mod_h323
# make install
```

2. 配置和加载模块

安装完毕后，如果一切正常，就可以在 FreeSWITCH 中加载该模块了：

```
freeswitch> load mod_h323
```

如果在加载时遇到类似下面这样的错误，可以在 Linux 命令行上执行“ldconfig”，然后重启 FreeSWITCH。

```
**libh323_linux_x86_64_.so.1.25.0: cannot open shared object file:
No such file or directory**
```

如果在加载模块时看到如下的错误，则说明 FreeSWITCH 默认没有安装相关的配置文件。

```
[ERR] mod_h323.cpp:483 open of h323.conf failed
```

可以在 mod_h323 源代码目录中找到 h323.conf.xml，然后复制到 conf/autoload_configs/ 目录中，如：

```
# cd src/mod/endpoint/mod_h323
# cp h323.conf.xml /usr/local/freeswitch/conf/autoload_configs/
```

然后再重新加载该模块就可以了：

```
freeswitch> load mod_h323
```

3. 呼叫测试

在正确加载完 mod_h323 以后，就可以进行拨打测试了。目前，该模块仅支持点对点的模式互通，并不支持 Gatekeeper。因此，可以在大部分的 H.323 客户端上直接呼叫 FreeSWITCH 的 IP 地址。呼叫到达 FreeSWITCH 后，将按照 h323.conf.xml 中的配置进行路由。当然，有些客户端允许在呼叫的 IP 地址前面加一个被叫号码，如可以尝试呼叫"1000@192.168.7.5"或"1000@192.168.7.5:1720"等。

如果在 FreeSWITCH 中使用 mod_h323 呼出，可以使用如下 Dialplan：

```
<action application="bridge" data="h323/6000@192.168.7.9"/>
```

其中，"h323"是该模块实现的一个呼叫字符串，后面的地址是欲呼叫的对端的 IP 地址。

14.6.2 mod_opal

mod_opal 的安装步骤与 mod_h323 类似，它也是需要一对相互匹配的 opal 库与 ptlib 库。在测试时，笔者是如下地址 http://sourceforge.net/projects/opalvoip/files/v3.12%20Eridani/Stable%206/ 下载的 ptlib-2.12.6.tar.bz2 以及 opal-3.12.6.tar.bz2。

ptlib 的安装方法与在 mod_h323 中的类似，在此就不多说了。只是注意，如果同时在一台主机上安装两个版本的 ptlib 可能会有冲突，如何在同一台主机上安装并运行不同版本的库文件超出了本书的范围，如果你对这些不熟悉的话，请分别在两台"干净"的机器上分别测试 mod_h323 与 mod_opal。

接下来，使用以下命令安装 Opal：

```
# tar xvjf opal-3.12.6.tar.bz2
# cd opal-3.12.6
# ./configure && make && make install
```

在 FreeSWITCH 源代码目录下使用如下命令安装 mod_opal：

```
# cd src/mod/endpoint/mod_opal
# make install
```

笔者在编译时遇到如下错误：

```
/home/app/work/freeswitch/src/mod/endpoints/mod_opal/mod_opal.cpp:1034:37:
error: 'class OpalConnection' has no member named 'SwitchT38'
```

错误的原因是 mod_opal 跟这里 Opal 的版本不兼容。不过，T38 是与传真相关的，这里，我们不测试传真，所以，可以在 mod_opal.h 文件中找到如下的行：

```
#define HAVE_T38 (OPAL_CHECK_VERSION(3,11,2) && OPAL_T38_CAPABILITY)
```

并将其修改为：

```
#define HAVE_T38 0
```

即暂时取消与 T38 相关的宏定义。然后就可以继续编译安装了。

安装完成后，在 FreeSWITCH 中加载：

```
freeswitch> load mod_opal
```

然后，如果有电话呼入，则可以在 FreeSWITCH 中看到日志。一旦呼叫到达了 Dialplan，就是我们的天下了。通过前面学过的知识，如何配置 Dialplan 对来话进行路由对我们来说已是小菜一碟了。因此，在此就不多介绍了。

另外，如果使用 mod_opal 呼出的话，它提供了 Opal 呼叫字符串，因而可以使用如下 Dialplan 进行呼出：

```
<action application="bridge" data="opal/h323:1000@192.168.7.9"/>
```

其中，"192.168.7.9" 是指对端的 IP 地址。

14.6.3 其他

上面我们重点讲了 H.323 中两个模块的安装，因为该过程的复杂性往往是读者学习和实验这两个模块的门槛。这里笔者是仅仅做了几个实验。如果读者想将这两个模块用于生产环境的话，还需要多测试一下，找一找这几个依赖库的最佳组合。

由于使用 H.323 的人比较少，因而大家对这两个模块开发的积极性也不高，因而它们的功能也不是很完善。如它们并不支持视频呼叫，也不支持使用 Gatekeeper 方式注册。不过，如果需要 Gatekeeper 的环境，也可以考虑与 GnuGK [⊖] 等专门的 H.323 Gatekeeper 配合部署，在此我们就不多讲了。

另外，笔者正在尝试重新写一个 H.323 模块——mod_ooh323，以支持视频通话。不过，该模块还没写好，暂时就不能写到书里了（等你看到本书时或许已经有了）。

14.7 小结

本章主要讲了使用不同的协议跟各种其他的设备进行对接。首先，本章通过 Dougango 和 IMS 的例子，帮读者扩充了 SIP 注册及对接的一些基础知识，并逐渐过渡到与 TDM 设备的对接上。

⊖ GnuGK 是一款开源的 H.323 Gatekeeper，请参考 http://www.gnugk.org/。

与 TDM 设备的对接，有板卡方案和网关方案。本章使用市场上常见的设备给出了实际的例子，并且用了很大的篇幅讲了 Sangoma 板卡及驱动的安装和配置。

后面，我们也讲到了与 Asterisk 的对接方法，并将 Asterisk 中的一些基本概念和模型与 FreeSWITCH 做了比较。

在本章最后，我们还讲了 FreeSWITCH 中 H.323 的实现、与 H.323 设备的对接，以及在编译安装时需要注意的问题。

一个人可能永远也不会用到这里讲的所有的设备或协议。但笔者希望读者能通过这些不同的例子，进行对比和思考，以对 FreeSWITCH 的运行方式、配置方法、设计理念、软硬件配合，以及通信设备的网络拓扑及部署方式有更好的理解。

第 15 章

其他技巧与实例

在前面的章节中，我们讲过一些基本的和比较高级一些的实例，涵盖了大部分常用的功能。但是，FreeSWITCH 功能非常多，还有一些常用的功能，如企业通信中常用的电话转接、代接等，我们还没来得及讲。另外，对于一些不常用的功能，以及生产环境中关心的压力测试、安全等话题，我们也将在本章简单介绍，以方便感兴趣的读者进一步研究。

15.1 转接和代接

来电转接和代接是企业 PBX 中的常用功能。来电转接分为盲转（Blind Transfer）和协商转（Attended Transfer）两种[⊖]，代接实现起来也很简单。下面我们分别加以介绍。

15.1.1 盲转

顾名思义，盲转就是将来电直接转到某一分机。该业务一般用于电话已经接听的情况，举例来说，转接的步骤一般是这样：A 呼叫 B，B 接听，A 与 B 通话，A 要求转 C，B 通过一个操作将来电转接到 C，C 开始振铃，B 挂断，C 接听，A 与 C 通话。

FreeSWITCH 默认的 Dialplan 实现了这种转接，方法是 B 在通话中按“*1”两个键，听到拨号音后，拨叫 C 的号码，B 挂机，A 与 C 通话。

这种转接魔术的根本原因在于下面这一行，在默认的 Dialplan 中的 Local_Extensions 中可以找到：

```
<action application="bind_meta_app" data="1 b s execute_extension::dx XML features"/>
```

首先，有电话打入后，如果是最终路由到一个内部分机，就会执行到这一行。至此，来话是 a-leg，现在还没有 b-leg。

bind_meta_app 是一个 App，它会在本次通话上绑定参数中指定的“1”这个按键，这个

⊖ 也有人分别叫一步转、两步转，或直接转移、出席转移等。

可以后续可由“*1”两个键激活。注意，上述代码中，“1”后面的“b”表示这个功能要绑定到 b-leg 上，虽然 b-leg 现在还没有。但是到有了以后，b-leg 可以按“*1”激活该项功能，而不是 a-leg。想想为什么[⊖]？

接着说，“b”后面的“s”表示什么呢？它表示，如果 b 按键激活该功能后，要在哪条腿上执行这个动作。这里的 s 表示 same，就是说，在哪条腿上接收到按键就在哪条腿上执行。

后面的“execute_extension”表示去 Dialplan 中找一个 extension 去执行一下，这里指到 XML features（它是一个 Context）中去找 dx 这个 extension。在默认的配置中可以在 features.xml 文件中找到。

总之，这一行执行完毕后只是相当于设了这么一个“套”，指出后面该怎么执行，直到 b-leg 按“*”才能激活该功能。接下来，Dialplan 最终会执行到后面的 bridge(user/B)，A 与 B 桥接并开始通话。

通话后，b-leg 就有了，它就是用户 B 所在的那条腿。在通话中，B 按下“*1”，就会执行到“dx”，我们到 features.xml 中去看具体的实现：

```
<extension name="dx">
  <condition field="destination_number" expression="^dx$">
    <action application="answer"/>
     <action application="read" data="11 11 'tone_stream://%(10000,0,350,440)'
digits 5000 #"/>
    <action application="execute_extension" data="is_transfer XML features"/>
  </condition>
</extension>
```

这里的 answer 只是保证电话是应答的状态。后面的 read 表示等待用户按键，也就是等待用户输入一个分机号（如 C 的号码）。参数“11 11”是自动收号位数，表示最少接收 11 位，最大接收 11 位。如果在收号近程中用户按了“#”（最后一个参数）号，则会立即停止接收（而不管最小和最大收号位数）。tone_stream 会产生一个拨号音，以提示 B 可以拨号了。B 拨完号之后，拨号的数据就存到了 digits 这个通道变量中。电话流程继续往下执行，接着又是 execute_extension，它的参数 is_transfer 是另一个 extension，我们可以在同一个文件中继续往下找到，其内容如下：

```
<extension name="is_transfer">
  <condition field="destination_number" expression="^is_transfer$"/>
  <condition field="${digits}" expression="^(\d+)$">
    <action application="transfer" data="-bleg ${digits} XML default"/>
    <anti-action application="eval" data="cancel transfer"/>
  </condition>
</extension>
```

可以看到，到了“is_transfer”以后，通过正则表达式“^(\d+)$”判断收到的数字是不是合法的号码，如果是，则执行 transfer App 进行转移。

⊖ 因为 aleg 可能是外部来话，你肯定不想外面的人乱按按键执行各种功能。

先说上述正则表达式不匹配的情况，即用户 B 输入的不是一个合法的分机号，那么 Dialplan 就会执行到“anti-action”一行，取消本次操作。最终结果就是——B 在听了一会拨号音并瞎按了半天后继续与 A 通话。

再说 B 按对了 C 号码的情况。执行到 transfer App 后，该 App 的参数中有一个“-bleg”，它什么意思呢？实际上，它的意思并不是绝对的 b-leg，而是一个相对的概念，即与当前那条腿桥接的另外一条腿。如果这么说还不明白，可以这样考虑：现在是 B 做了这么多操作，因而这些操作都是在原来的 b-leg 上执行的。而 transfer 这里是要把 A 转走，由于当前的是 b-leg，再加上“-bleg”后就相当于“负负得正”，因而意思是把 a-leg 转走。转到哪儿呢？转到“digits”变量所代表的值，即刚才 B 输入的 C 的号码。

这时 C 开始振铃，如果 C 接听，A 就可以与 C 通话了。同时，A 被转走后，B 就没事干了，电话自己就挂掉了。

当然，上面绕了这么一大圈才完成这个转接功能，看起来挺复杂的。实际上，聪明的读者也许已经注意到了，在“dx”那个 extension 中有这么一句注释：

```
<!-- In call Transfer for phones without a transfer button -->
```

它的意思是，该功能是给那些没有 Transfer 键的话机用的（也就是一般的模拟话机，由于没有 Transfer 按键，因此一般需要靠 DTMF 按键执行这些功能）。对于 SIP 话机而言，通常都会有一个 Transfer 键，一按就转了，很方便。那么这是怎么实现的呢？答案是 SIP REFER。

REFER 是在 RFC3515 ⊖中定义的，它规定了 SIP 中电话转接的几种实现方式，具体协议流程的实现跟具体话机终端有关，图 15-1 描述了一种典型的实现方式。

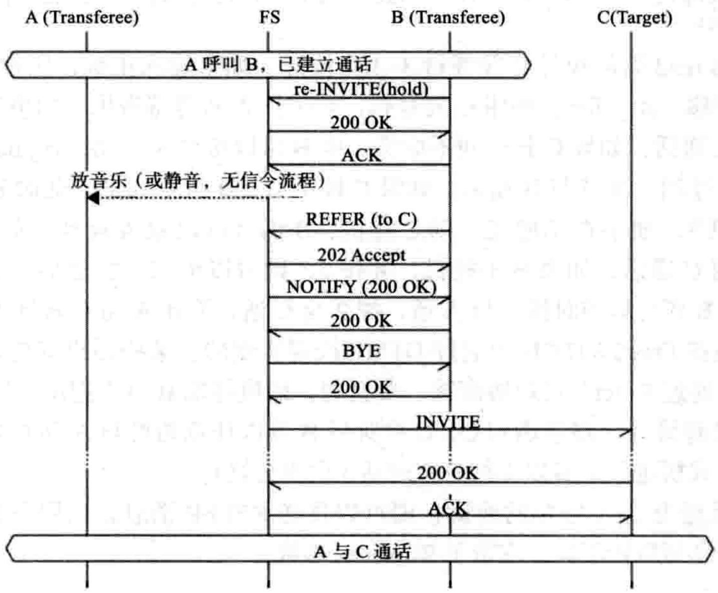


图 15-1　使用 Refer 进行呼叫转接

⊖ 参见 www.ietf.org/rfc/rfc3515.txt。

首先A与B已建立通话，这时候B想把A转接给C。这里B称为Transferor，它是转接的发起者；而A称为Transferee，它是被转接的一方；C称为Target，是转接的目的地。转接成功后A与C通话。

B首先发re-INVITE请求给FS(FreeSWITCH)，请求将B的电话置为Hold(保持)状态，FS收到请求后就给A播放保持音乐。同时，B的话机放拨号音，以提示用户输入被叫号码。B输入C的号码后，B给FS发REFER请求。FS收到后会释放B，并同时呼叫C。如果C正常接听，则A与C通话，转接完成。

15.1.2 协商转

读者读到这里可能会发现一个问题：在上述盲转的情况下，如果C长时间不接听（久叫不应）或C占线，则转接会失败，A的电话会被挂断。A可能会重新呼叫B要求转接，这对A的体验是很不好的。

FreeSWITCH通过att_xfer这App支持协商转。在默认的拨号计划中也是可以实现的，与盲转不同的是，在进行转移时，B按"*4"激活"att_xfer"功能，配置如下：

```
<action application="bind_meta_app" data="4 b s execute_extension::att_xfer XML features"/>
```

同样，我们在features.xml中找到"att_xfer"条件：

```
<condition field="destination_number" expression="^att_xfer$">
  <action application="read" data="3 4 'tone_stream://%(10000,0,350,440)' digits 30000 #"/>
  <action application="set" data="origination_cancel_key=#"/>
  <action application="att_xfer" data="user/${digits}@$${domain}"/>
</condition>
```

它照样使用read播放拨号音并等待3~4位按键，如果输入正确，则使用att_xfer这个App处理呼叫转移。att_xfer会呼叫C的号码，同时让A听等待音乐。如果呼叫C失败，则B仍然可以与A通话；如果C长时间不应答，则B可以按"#"（由origination_cancel_key设置）号键取消呼叫，继续与B通话；如果C接听后，B与C通话，此时B可以询问C是否愿意接听电话[⊖]，如果C不愿意，则C挂机，B仍然可以跟A通话；如果C接受通话，则B挂机，A与C通话；如果B不挂机，并按3，则可以形成三方通话，大家一起说。还有更有意思的，B还可以随时按1与A通，按2与C通，而让A与C永远不通……

上述功能是在FreeSWITCH中通过DTMF按键实现的。某些话机支持多路通话，因而可以在话机端（通过Refer）实现协商转。典型的，话机终端B可以把第一路电话置于Hold状态，然后再发起另外一路通话到C，C接听后B可以任意切换与A和C之间的通话，并可以通过本地会议桥进行混音以支持三方通话（也叫会议）。

此时B如果想退出A与C的通话，则可以发送REFER消息，让服务器把通话中的B替换为C。该消息与盲转不同，它带了Replaces参数：

⊖ B与C协商，这是也协商转的来源。

```
    Refer-To: sip:1002@192.168.1.118?Replaces=1388923627@192.168.1.110;to-tag=
NDj261X80jpKF;from-tag=1013380895>
```

为了阅读方便，上面的消息是经过“urldecode”后的，实际的消息内容是用“urlencode”编码的字符串。它是这样产生的：A（1000）呼叫 B（1004），此处 B 是亿联话机，B 接听后，按下话机上的 Conf 软键（代表 Conference，会议）然后呼叫 C（1002）。C 接听后，B 与 C 通话。B 可以通过话机上的 Swap 按钮进行切换并与 A 或与 C 通话，也可以再次按 Conf 键启用本地混音实现 A、B、C 三方的电话会议。与我们上面讲的例子不同，上面的例子三方通话是由 att_xfer 实现的，参与方只有三方，即三个 Channel；而本例中是 4 个 Channel，因为 B 话机同时有两个 Channel 连接到 FreeSWITCH。

如果 B 想退出会议，但保持 A 与 C 的通话，则 B 发送 REFER 消息让 FreeSWITCH 把它替换掉，即让 FreeSWITCH 把 A 与 C 桥接起来，释放掉与 B 的两路通话。

完整的 REFER 消息如下：

```
    ------------------------------------------------------------------------
    recv 594 bytes from udp/[192.168.1.110]:5063 at 05:59:43.368543:
    ------------------------------------------------------------------------
    REFER sip:mod_sofia@192.168.1.118:5060 SIP/2.0
    Via: SIP/2.0/UDP 192.168.1.110:5063;branch=z9hG4bK1882546254
    From: <sip:1004@192.168.1.110:5063>;tag=485693841
    To: "Extension 1000" <sip:1000@192.168.1.118>;tag=H9cZZNttcFX8g
    Call-ID: affaa293-566c-1231-e2ba-79a3b2b753b0
    CSeq: 45647480 REFER
    Contact: <sip:1004@192.168.1.110:5063>
    Max-Forwards: 70
    User-Agent: Yealink SIP-T22P 7.70.0.140
    Refer-To: <sip:1002@192.168.1.118?Replaces=1388923627%40192.168.1.110%3Bto-
tag%3DNDj261X80jpKF%3Bfrom-tag%3D1013380895>
    Referred-By: <sip:1004@192.168.1.110:5063>
    Event: refer
    Content-Length: 0
```

有兴趣的读者可以自行跟踪 SIP 消息看一下，详细的流程（及流程图）就不赘述了。

15.1.3 代接

代接是指（别人给 A 打电话时）A 电话振铃后，在 B 话机上进行接听（代替 A 来接听）。一般用于办公室中某工位上没人其他工位上的人代为接听的场景。

FreeSWITCH 默认的 Dialplan 中就有与代接相关的例子。其中，886 为全局代接。即当有分机振铃时，在另外的话机上直接按 886 就能接听，同时原先振铃的话机结束振铃。“*8”为组内代接，也就是同组代答，即在上述情况下按“*8”只能代接本组内的正在振铃的分机。以上两种方式在有多个分机同时振铃时只能接听最后振铃的那一个。此外，还有一个“**”前缀码，拨打“**”加上指定的分机号就能直接代接指定分机，如拨“**1001”就可以接听正在振铃的 1001 分机上的电话。

代接是使用 intercept App 实现的，具体的 Dialplan 我们在此就不详细解释了，有兴趣的

读者可以参考默认的 Dialplan（dialplan/default.xml）中的相关内容（搜索 intercept），结合我们本书中讲过的知识是很容易理解的。

15.2 共享线路呈现

共享线路呈现（Shared Lines Appearence，SLA）也是企业应用中一个非常有用的功能。该功能在模拟话机中是没有的，只有 SIP 话机才能实现该功能。使用该功能可以在自己的话机上监视其他话机的状态，从而知道另一个电话是否处于忙或闲的状态。比如说在老板 - 秘书的场景中，如果有人打秘书的电话想找老板，而秘书转电话时碰巧老板的电话正在占线，就会导致转接不成功，耽误时间。而如果秘书事先知道老板的电话是否在忙，就可以直接判断是否要将电话转给老板，或者告诉主叫用户先等一会。FreeSWITCH 就支持这种功能。在使用时应确认 Profile 中的设置开启了以下两项：

```
<param name="manage-presence" value="true"/>
<param name="manage-shared-appearance" value="true"/>
```

下面以 Seven Du（607）和 Sonic Gao（608）为例。607 使用 XLite 注册。注册完毕后通过菜单项“Contacts”→“Add Contact”添加一个联系人 608，并确保开启了 Presence 功能，如图 15-2 所示。

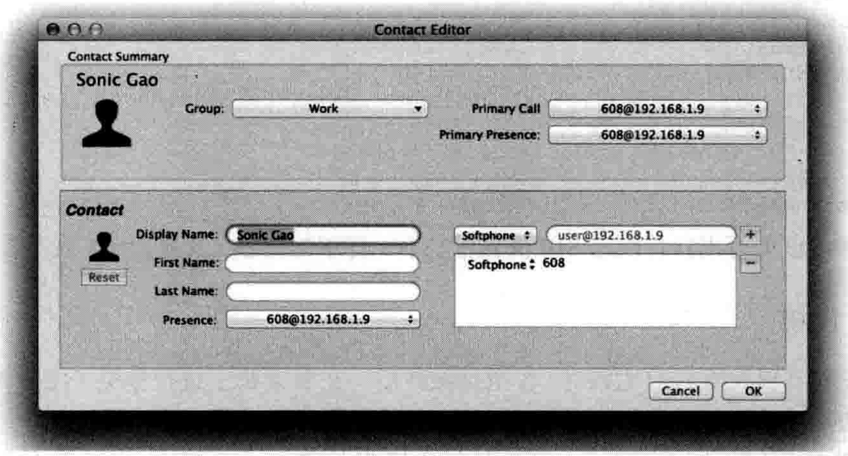
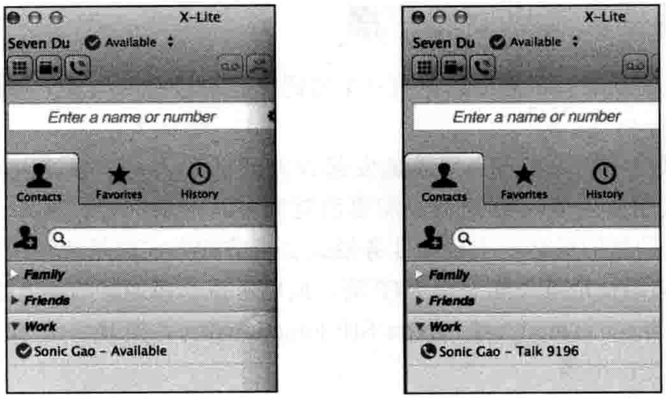


图 15-2 在 XLite 中添加 Presence 联系人

添加完成后，主界面上就能显示联系人的状态了，如图 15-3a 所示，现在 608 处于注册状态，显示是绿色的 Available 状态（见图 15-3a）。如果 608 正在通话，则会变为红色的 Talk 状态，图 15-3b 显示 608（Sonic Gao）正在跟 9196 通话。

账号 608 是从一个亿联话机上注册的，该话机也支持线路共享功能。打开话机的 Web 设置界面，依次顺着菜单走到“话机设置”→“可编程按键”→“账号键”，如图 15-4 所示。



a)　　b)

图 15-3　XLite 上的联系人状态

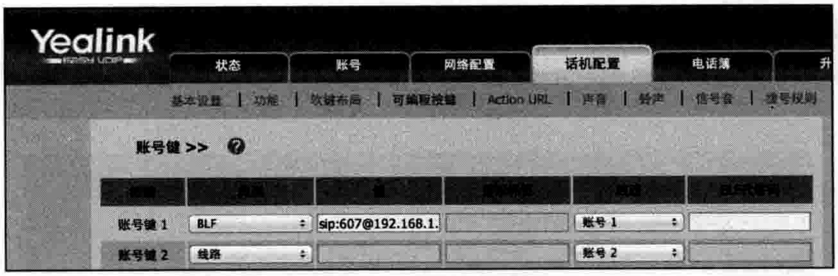


图 15-4　在亿联话机上配置 SLA

在图 15-4 中所示的页面上，选择账号键“1”，把类型由“线路”改为 SLA，“值”里输入你想监视的话机账号，这里是“ sip:607@192.168.1.9”，在注册“线路”一栏也选择账号 1，保存设置后，话机上账号 1 对应的灯就会常亮，表示 607 是正常空闲的。如果 607 正在通话，该灯就会闪烁。如果 15-5 所示，话机上第一个灯是常亮或闪烁。



图 15-5　话机灯的状态

15.3 使用组播功能做网络广播

在某些特殊的场景下，拿起电话拨打一个号码，就可以对一大群人喊话，广播功能可以大大提高工作效率。

广播功能可以使用会议实现，简单地发起 *N* 路通话加入一个会议也可以做到广播的效果。不过，那样实现要建立 *N* 路通话，需要消耗很多的网络资源；另外，也无法保证对方能及时接听，影响信息的送达。实现该业务模式最经济的方式就是使用组播（Multicast，或称多播）。组播只向组播地址发送一个 RTP 流，而监听该组播地址的所有主机就都能收到。

FreeSWITCH 有一个 mod_esf（Extra SIP Functionality）模块，它提供一个 esf_page_group 的 App 可以支持组播。

esf_page_group 有三个参数，分别是：

- ❑ 组播地址，默认为 224.168.168.168。
- ❑ 端口号，默认为 35467。
- ❑ 控制端口号，默认为 6001。

要能收到组播包并播放声音，也需要配置亿联话机以启用这项功能⊖，配置界面如图 15-6 所示。



图 15-6 配置亿联话机接受组播包

该配置项比较难找，依次找到菜单“电话簿”→“组播”，在第一个 IP 地址框里填入 224.168.168.168:34567，标签栏可以任意填，起一个好记的名字就行。

在 FreeSWITCH 默认的配置中，拨打号码 7243 就直接向该地址发送组播，所以现在你可以拿电话拨打 7243 试一下了。

默认的 Dialplan 配置如下：

```
<extension name="rtp_multicast_page">
  <condition field="destination_number" expression="^pagegroup$|^7243$">
    <action application="answer"/>
    <action application="esf_page_group"/>
  </condition>
</extension>
```

⊖ 笔者不知道其他话机是否具有这样的功能，最初亿联也没有，是笔者帮助 FreeSWITCH 的作者 Anthony Minessale 联系到亿联的工程师后他们才实现的。

如果你想发到其他的地址，可以配置相关参数，如下列配置可以将 RTP 包发到组播地址 224.0.0.100：

```
<action application="esf_page_group" data="esf_page_group 224.0.0.100 34567 6001"/>
```

与普通的 IP 地址不同，组播需要配置组播地址。众所周知，在 IPv4 中，组播地址的范围是从 224.0.0.0 到 239.255.255.255，由于实际用到组播的业务却很少，因而好多人可能不是很熟悉。在有的系统上需要配置组播路由，如以下命令可以在 Linux 系统的 eth0 上配置组播路由：

```
ip route add 224.0.0.0/4 dev eth0 src 192.168.5.2
```

使用上述命令还需要 iproute2 软件包支持。另外，在实际应用中为避免组播风暴及潜在的冲突，可能需要把具有组播功能的话机都划分到一个 VLAN 上。再者，组播包一般不能穿越路由器，如果要跨路由器组播的话，需要支持组播的路由器，并进行适当的配置，这些都超出了本书的范围，有兴趣的读者可以自行参考相关资料。

15.4 DTMF

DTMF（Double Tone Multiple Frequecy，双音多频）是一种通话过程中的号码传输方式，特别是在 IVR 类的应用中，一般的电话菜单都是通过按键控制的。在传统的 TDM 电话网络中，如果通话中的用户按下了话机上的一个键，话机就会产生一个 DTMF 的音频信号送到远端的交换机上，交换机会将该信号送到对端的用户那里，如果对端也是一个用户，则他就能听到一些“嘀嘀”的按键音；如果对端是一个 IVR，它就能检测这种信号，并还原成用户按键的数字，进而知道用户选择了哪个菜单。

在 SIP 通话中，传输按键信息的方式相对较多。虽然有些信息可以完全不用双音多频的信号来表示，但沿用传统的叫法，我们仍然把按键信息称为 DTMF。

15.4.1 带内 DTMF

在基于 SIP/RTP 的通话中，如果跟传统的 TDM 电话对接，直接把 TDM 电话设备产生的双音频信号按与音频编码同样的方式进行编码并放到 RTP 数据中传输，则这种 DTMF 称为带内 DTMF（Inband）。FreeSWITCH 默认不使用带内 DTMF。因为，检测带内 DTMF 必须要实时检测 RTP 包中的内容，会多耗费很多 CPU。不过，传统的 TDM 设备只支持这种方式，因而如果要跟传统的 PSTN 网络的交换机进行 SIP 对接，则往往默认会使用这种方式。由于 PSTN 网络上的交换机一般不会使用到 DTMF，它们只需要对 DTMF 信号进行简单的透传，而不需要进行深入检测，因而对它们来说无所谓是否耗费资源。

如果在 FreeSWITCH 中使用带内 DTMF，则需要在对应的 Profile 中（如“internal”或“external”）中设置如下参数：

```
<param name="dtmf-type" value="inband"/>
```

或者在通话时，在 Dialplan 中通过设置如下的通道变量实现：

```
<action application="set" data="dtmf_type=inband"/>
```

当然，设置了上述变量并不能使 FreeSWITCH 检测到带内的 DTMF，还必须在 Dialplan 中明确使用 start_dtmf 这个 App 打开 DTMF 检测。比如，对于来话而言，就需可以使用如下的 Dialplan：

```
<action application="set" data="dtmf_type=inband"/>
<action application="start_dtmf"/>
<action application="ivr" data="some_ivr"/>
```

对于去话，则可以通过设置一个钩子通道变量[⊖] "execute_on_answer" 实现，如：

```
<action application="export" data="nolocal:execute_on_answer=start_dtmf"/>
<action application="bridge"data="sofia/gateway/pstn/${destination_number}"/>
```

在上述 Dialplan 中，我们使用 "export" 设置了一个变量 "execute_on_answer"，该变量是以 "nolocal:" 开头的，因此，它将只在 b-leg 上生效。在执行到 "bridge" 时，将产生 b-leg。本例中的 b-leg 通过一个 pstn 网关呼叫被叫号码，当对端接听后，FreeSWITCH 就会执行 execute_on_answer 钩子变量所指定的 App，即 start_dtmf，它用于开启带内 DTMF 检测。

当然，如果对 export" 这个 App 以及其原理比较熟悉的话（参见 6.1.9 节），便知道上述 Dialplan 跟如下的写法是等价的：

```
<action application="bridge"
data="{execute_on_answer=start_dtmf}sofia/gateway/pstn/${destination_number}"/>
```

15.4.2 RFC2833

RFC2833 [⊖]中定义了在 RTP 中传输 DTMF 信息的另一种方式。该方式通过特殊的 RTP Payload 进行标志，标志出包含 DTMF 信息的 RTP 包，因而不需要对所有的 RTP 包进行深度的频率检测就可以很容易认出包含 DTMF 信息的包，而这种包里的 DTMF 信息是以文本表示的，因而也不需要计算。与 Inband 检测相比，RFC2833 方式大大节省了 CPU。这种方式是现行的 VoIP 通话中比较常用的方式，也是 FreeSWITCH 中默认的方式。

RTP 包中 DTMF 的 Payload Type 值是在 SDP 消息中使用 "telephone-event" 协商的，如下面的 SDP 消息中，表示 Payload Type 为 101 的 RTP 包为包含 DTMF 消息的包。

```
v=0
o=FreeSWITCH 1371848118 1371848119 IN IP4 192.168.1.9
s=FreeSWITCH
c=IN IP4 192.168.1.9
t=0 0
m=audio 31988 RTP/AVP 8 101
a=rtpmap:8 PCMA/8000
```

⊖ FreeSWITCH 中有一系列的以 execute_on_ 开头的通道变量，这些变量在系统中设置一些钩子，当 Channel 执行到一定状态时，便会执行这些钩子指定的 App，比如此处的 execute_on_answer 就会在相关的 Channel 被应答时执行指定的 App。

⊖ 参见 http://tools.ietf.org/html/rfc6455。

```
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-16
a=silenceSupp:off - - - -
a=ptime:20
m=video 19008 RTP/AVP 123
a=rtpmap:123 H264/90000
```

如果在媒体协商时看不到“telephone-event”，则证明它不支持 DTMF2833 模式。

注意

由于 RTP 底层使用了不可靠的 UDP 传输协议，因此 RFC2833 规定包含同一 DTMF 信息的包要按一定的规则重发，以减少可能的丢包带来的影响。而且，DTMF 的 RTP 包跟音频 RTP 包是在一个媒体流（相同的“IP 地址：端口号”）中传输的，因此包含 DTMF 的 RTP 包中的时间戳以及 Marker 标志还需要跟普通音频正确配合才能使用。这就带来了该协议的一些复杂性。比较令人遗憾的是，有很多设备的 DTMF2833 实现都有 Bug，而且几年甚至十几年都不改。为了能够跟这些“顽固”的设备对接，FreeSWITCH 内部也进行了相应的妥协，针对某些设备提供了一些专门的配置参数。当然，如何使用这些具体的参数超出了本书的范围，如果在与这样的设备对接时发现不符合 RFC2833 规定的实现，可以查阅有关资料尝试在 FreeSWITCH 中设置相关参数进行解决㊀。

15.4.3 SIP INFO

DTMF 也可以在 SIP 的 INFO 消息中传送。INFO 消息跟一路通话有关，用于通话中的双方传送一些消息，其中一种消息就是传输 DTMF 消息。

使用 SIP INFO 消息甚至比使用 RFC2833 更有效，但是，有一个缺点是由于 SIP 消息跟用于传送声音的 RTP 流使用不同的 UDP IP 地址和端口，在实际传输中可以经过完全不同的路由设备，因而用 SIP INFO 传送的 DTMF 可能会与实际听到的声音不同步。

如果在 SDP 协商时没有找到“telephone-event”（即不能使用 RFC2833），则 FreeSWITCH 就会默认启用 INFO 来收发 DTMF。当然，也可以在 Profile 中使用如下参数明确指定使用 INFO 来传送 DTMF：

```
<param name="dtmf-type" value="info"/>
```

或者在 Dialplan 中针对某一路通话决定是否使用 INFO：

```
<action application="set" data="dtmf_type=info"/>
```

关于 DTMF 的知识我们就介绍这么多，请读者根据自己的需要多加练习。

15.5 号码连选

虽然有些运营商也开始尝试开放 SIP 号码，但大部分还是谨慎的、试探性的。因

㊀ 参见 http://wiki.freeswitch.org/wiki/RTP_Issues。

此一般不提供 SIP 对开中继的方式，而是开放单个的接入号码。虽然这不是理想的方式，但有总比没有好。在此，假设我们从运营商那里获得 10 个 SIP 账号，号码范围是 xxxxxx30 ~ xxxxxx39。我们来看一下如何有效地使用这些号码。

15.5.1 注册到运营商服务器

我们可以在 FreeSWITCH 中添加一些网关，以便注册到运营商的 SIP 服务器上去（应该是一个 SBC）。

网关的配置文件如下，为了使用方便，我们让网关名称（name）的后两位与号码的最后两位相同：

```
<gateway name="gw30">
    <param name="realm" value="218.56.x.x"/>
    <param name="username" value="xxxxxx30"/>
    <param name="password" value="xxxx"/>
    <param name="register" value="true"/>
</gateway>

<gateway name="yt31">
    <param name="realm" value="218.56.x.x"/>
    <param name="username" value="xxxxxx31"/>
    <param name="password" value="xxxx"/>
    <param name="register" value="true"/>
</gateway>

......
```

上面我们仅列出了两个网关账号的配置，其他账号依此类推。

15.5.2 通过单个号码呼出

15.5.1 节配置的网关注册成功后，我们就可以通过这个号码（又称为线路）打入打出电话了。我们可以使用如下命令快速试一下是否能通过某一条线路（如 gw30）成功呼出：

```
freeswitch> originate sofia/gateway/gw30/1860535xxxx &echo
```

测试成功后，就可以设置如下的 Dialplan 让所有分机都可以通过该网关呼出了：

```
    <extension name="Outbound Call">
  <condition field="destination_number" expression="^(1[358].*)$">
    <action application="bridge" data="sofia/gateway/gw30/$1"/>
  </condition>
</extension>
```

注意

出于简单起见，本例中使用的正则表达式仅匹配手机号。上述配置将统一使用网关 gw3Ø（即通过号码 xxxxxx3Ø）呼出。很容易想到，如果有第二个人同时进行呼叫时，由于该网关对应的号码占线，会导致呼叫失败。

15.5.3 使用随机数做号码连选

为了能自动选择一个网关呼出，我们想办法从这 10 个网关中自动选择一个进行呼出。这种选择的过程就称为选线，也称为号码连选。当然，号码连选最简单的实现方法是使用一个随机数。见下面的 Dialplan：

```
<action application="set" data="gw=gw${expr(randomize(&x);ceil(random(30,39,&x)))"/>
<action application="bridge" data="sofia/gateway/${gw}/$1"/>
```

其中，“expr”是一个 API，我们用它的 randomize 方法产生一个从 30 到 39 之间的随机数（如 33），在该随机数前面加上“gw”字符（变为 gw33），并把它赋值给一个“gw”通道变量（使用“set”）实现。有了该通道变量后，在“bridge”的参数中就可以使用“${gw}”引用该变量（在本例中它的值就是 gw33），实现动态选择一个随机的网关。

当然，这种选线算法有一个缺点，就是它不记录实际号码的忙闲状态，如果选到正在通话的号码时，通话还是会失败。通过下面的方式，我们可以做一个改进的算法：

```
<action application="set"
    data="gw1=gw${expr(randomize(&x);ceil(random(30,39,&x)))"/>
<action application="set"
    data="gw2=gw${expr(randomize(&x);ceil(random(30,39,&x)))"/>
<action application="bridge"
    data="sofia/gateway/${gw1}/$1|sofia/gateway/${gw2}/$1"/>
```

该方法的思路是，同时选择两个网关（gw1 和 gw2），如果一个失败，则走另一个（该方法不能从根本上解决选到正在通话的线路问题，但可以大大减少失败的概率）。

15.5.4 使用 mod_distributor 进行连选

除随机数外，FreeSWITCH 也提供了一个 mod_distributor 模块模块，专门实现这种连选功能。首先，要安装该模块，进入 FreeSWITCH 的源代码目录，执行如下命令：

```
make mod_distributor-install
```

然后，在 conf/autoload_configs/distributor.conf.xml 中进行如下设置：

```
<list name="dist1" total-weight="10">
<node name="30" weight="1"/>
<node name="31" weight="1"/>
<node name="32" weight="1"/>
<node name="33" weight="1"/>
<node name="34" weight="1"/>
<node name="35" weight="1"/>
<node name="36" weight="1"/>
<node name="37" weight="1"/>
<node name="38" weight="1"/>
<node name="39" weight="1"/>
</list>
```

其中，我们配置了一个列表（list），它的名字是“dist1”，总体的权重（total-weight）是“10”。该列表有好多节点（node）组成，其中每个节点的权重（weight）为“1”[⊖]。可以看

⊖ 如果某条线路支持多线呼出，我们也可以增加它的权重。

出，这些节点的名字跟我们线路号码的最后两位相同。

FreeSWITCH 加载该模块后，我们就可以先用如下的命令进行一下测试：

```
freeswitch> distributor dist1
32
freeswitch> distributor dist1
35
```

distributor 是该模块提供的一个 API 命令，它可以用于从预定义的列表中根据权重选择一个节点，并返回该节点的名称。然后，我们就可以在 Dialplan 中使用它来帮助我们选线了：

```
<extension name="gw">
  <condition field="destination_number" expression="^(01[358].*)$">
    <action application="bridge"data="sofia/gateway/gw${distributor(dist1)}/$1"
loop="2"/>
  </condition>
</extension>
```

其中，我们使用 ${distributor(dist1)} 让 mod_distributor 帮我们选择一个节点，并使用它作为网关的名字向外呼出。当然，与上一节使用的随机数的方案相比，它也聪明不了许多，因为，该模块也没有记录线路是空闲还是忙的。因此，我们使用了 Dialplan Action 中的 loop 属性，如果第一次呼叫失败，它将再试一次（重新执行该 Action）。

当然，与上一节的随机数方案比起来，它还是聪明一点的。例如，虽然我们配置了 10 个网关，但并不一定所有时间所有的网关都能正常注册上。通过如下方法，就可以让 distributor 在生成选择节点时，排除掉处于 down 状态（即不可用状态）的网关：

```
<action application="bridge"
data="sofia/gateway/gw${distributor(dist1 ${sofia(profile internal gwlist down)})}/$1"/>
```

15.5.5 其他

上述的几种方案在一般话务量不是很高的情况下，问题应该不大。不过在很繁忙的场合（如线路占用率 90% 以上），可能就会出现较高的呼损（即呼叫失败，我们这里更多的是指在还有空闲线路的情况下的呼叫失败）。因此，为了提高线路的利用率，降低呼损，我们需要更好的算法。

当然，没有现成的方案来解决该问题，这里我们仅探讨一下实现的思路。首先，实现该算法的时候，应该能检测并记住线路的状态，避免选到坏线（如网关注册失败的线路）或忙线；其次，应该有冲突解决方案，即能够解决同抢的情况（如刚刚选择了一个空间闲的线路，但在呼出前的瞬间忽然来了一通呼入的电话）。

当然，实现这种算法比较复杂，而且如果实现不好，也不一定能减少呼损。所以有时候，多开一些网关线路，利用减少线路利用率来换取更低的呼损，选择一个折中的值还是可以接受的。

另外，在这些网关线路也会有大量呼入的情况下，呼损率肯定会更高。这时候，可以采取将呼入和呼出分开的方法，即一部分线路只允许呼入，另一部分仅用于呼出（最好能在运

营商的交换机上做限制，让该号码永远呼叫不进来）。具体两部分的比例应该根据实际情况设置，当然使用这种办法也会更进一步降低线路利用率。

15.6 收发传真

无论社会如何发展，老技术永远有它特殊的生命力，生生不息。传真技术也是如此。在互联网技术飞速发展的今天，电子邮件、即时通信以及基于移动互联网的各种应用，如微博、微信等，铺天盖地地发展，但我们在企业应用中，还是离不开传真机。当然，现代的传真机大多数都集打印、扫描等功能于一体了，但还是使用 TDM 通信的方式收发传真。

为了将这些老旧的技术及设备融入 SIP 世界里，很多人也进行了各种努力，因而我们也可以非常方便地使用它。连接 TDM 传真机最简单的方式是使用一个模拟转 SIP 的网关，将它变成 SIP 后在 FreeSWITCH 中收发传真就变得容易了。如我们可以使用如下命令，呼叫一个号码，并开始收传真。

```
freeswitch> originate sofia/gateway/gw1/xxxx &rxfax(/tmp/test.tiff)
```

传真功能是在 mod_spandsp[1]中实现的。它实现了一个 rxfax 的 App 用于收传真。普通的传真机是这样工作的，A 呼叫 B，首先建立正常的通话，电话打通以后 A 告诉 B 它想发一个传真，因此 B 会按下一个按钮将传真机切换到传真模式，这时 A 这一端将听到“吱吱”的传真音，然后 A 端按下传真机上的“发送键”（或传真开始键）发送传真。

有 SIP 参与的传真过程也是类似的。它们首先先建立正常的 SIP 连接，然后如果一方想发传真或收传真，则它的终端就会给另一方发送 SIP“ re-INVITE”消息，与对方协商将 RTP 媒体流切换到 T.38（或 T.30）传真图像模式，协商完成后开始发送传真。

上面我们讲到的“ rxfax”是收传真的 App，它会将接收到的传真存到本地的一个 TIFF [2]格式的文件中。

发送传真与与此类似，只是需要预先将欲发送的内容转成 TIFF 格式的文件。如下命令可以呼叫一个号码并发送传真：

```
freeswitch> originate sofia/gateway/gw1/xxxx &txfax(/tmp/test.tiff)
```

可以使用 Gostscript 或 Imagemagick 图像处理工具将欲发送的传真内容从原来的格式（如 PDF）转换成 TIFF 格式。如，下列 gs（即 Gostscript）命令可以将 fax.pdf 转换成 fax.tiff：

```
gs -q -r204x98 -g1728x1078 -dNOPAUSE -dBATCH -dSAFER -sDEVICE=tiffg3
-sOutputFile=fax.tiff -- fax.pdf
```

注意，其中“204x98”表示分辨率。传真文件的分辨率就是比较奇怪——横向和纵向的分辨率是不同的。另外，传真页面也需要有特定的大小，大部分传真机都能接收页面大小为

[1] 基于 Spandsp 库，当然除了传真之外它还实现了一些语音编解码。参考 http://wiki.freeswitch.org/wiki/Mod_spandsp。

[2] 传真常用的格式，与常见的 JPEG 或 PNG 格式不同，但类似于 PDF，它主要的特点是支持多页，而这一点在传真文件中非常重要。

"1728x1078"的传真，其他尺寸的因传真机而异。

上面我们只讨论了收发两端均是 SIP 的情况。如果在 FreeSWITCH 内部也使用如 ftdm 这样的 Endpoint（配合模拟或数字板卡），那么 FreeSWITCH 也可以通过 t38gateway 这个 App 进行传真媒体的转换㊀。

最后，我们再来看一下默认的 Dialplan 的配置。配置的 Dialplan 配置了两个 Extension，"9178"用于收传真，"9179"用于发传真。具体内容如下：

```
<extension name="fax_receive">
  <condition field="destination_number" expression="^9178$">
    <action application="answer" />
    <action application="playback" data="silence_stream://2000"/>
    <action application="rxfax" data="/tmp/rxfax.tif"/>
    <action application="hangup"/>
  </condition>
</extension>

<extension name="fax_transmit">
  <condition field="destination_number" expression="^9179$">
    <action application="txfax" data="/tmp/txfax.tif"/>
    <action application="hangup"/>
  </condition>
</extension>
```

读者可以自行拨打测试一下。Dialplan 的内容很简单，我们就不详细解释了。

15.7 多租户

很多人将 FreeSWITCH 用于云计算平台，而 VoIP 云计算除了要支持大规模的并发呼叫外，更重要的是要支持多租户㊁技术。简单来讲，多租户就是在一个系统中（或更简单点，一台 FreeSWITCH 服务器上），支持多个彼此相互独立的 PBX（如属于不同公司的）应用，这些不同的 PBX 中可能有相同的分机号，而不会产生冲突。

FreeSWITCH 在设计之初就考虑到了这一点，它是用 Domain 实现的。

15.7.1 Domain 简介

说起 Domain，大家一般都会联想到域名，或者更精确一点说，联想到一个 FQDN㊂。但 FreeSWITCH 中的 Domain 是指一个域，或者在这里我们说到多租户的时候，可以说它指一个租户。实际上，它就是唯一标志一个域的字符串。当然，由于它可以是任意的字符串，那么用实际的域名和 IP 地址也是没有问题的。但是，要记住，虽然你可以使用域名或实际的 IP 地址作为 Domain，但并不表示 Domain 跟域名或 IP 地址以及 DNS 有任何关系，

㊀ 详情参考 http://wiki.freeswitch.org/wiki/Mod_spandsp。

㊁ Multitenancy。参见 http://en.wikipedia.org/wiki/Multitenancy。

㊂ Fully qualified domain name，译作完全资格域名，或绝对域名。参见 http://en.wikipedia.org/wiki/Fully_qualified_domain_name。

FreeSWITCH 也永远不会把 Domain 通过 DNS 解析成域名，反之亦然。

讲到这里，读者可能听得一头雾水，那么为什么把问题搞这么复杂呢？答案很简单，为了支持多租户。

我们首先来看一下 conf/vars.xml 中如下的配置：

```
<X-PRE-PROCESS cmd="set" data="domain=$${local_ip_v4}"/>
<X-PRE-PROCESS cmd="set" data="domain_name=$${domain}"/>
```

其中，上述配置配置了两个全局变量 domain 和 domain_name，前者用作一个核心变量——当系统需要一个 Domain 值，而通过当前的环境又找不到这么一个值时，就用该变量的值顶替；而后者用作一个 Dialplan 中的变量，它只是跟每一通电话相关的。当然默认两者的值都等于 local_ip_v4 的值。而 local_ip_v4 是一个动态计算出的变量值，它一般是当前能上网的网卡的 IP 地址（在取不到的情况下可能为 127.0.0.1）。通过将 local_ip_v4 的值设成两个变量的默认值是因为这样的话在默认安装后不需要任何配置就可以在所有环境中都正常使用。

为了说明 Domain 跟实际的 FQDN 域名没有任何关系，我们来做如下实验。首先，如图 15-7 所示，在 Domain 中填入“dujinfang.com”。注册时选择“Send outbound via Proxy”，并在 Proxy 中输入 FreeSWITCH 的 IP 地址。在本例中，笔者试验的客户端和服务器的 IP 地址都是 192.168.7.5。

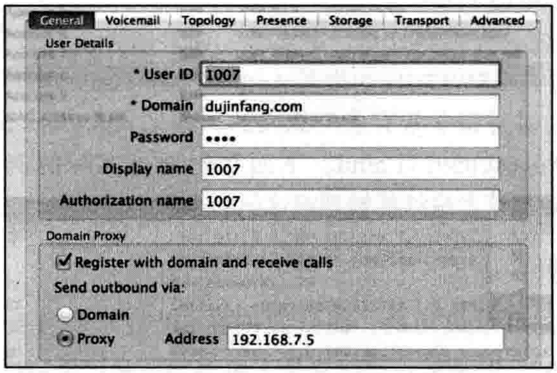


图 15-7　通过 Proxy 注册

经过这样配置后，我们在 FreeSWITCH 中开启 SIP trace 可以看到如下的注册消息（为了方便讲解我们加了行号）：

```
01 ----------------------------------------------------------------
02 recv 789 bytes from udp/[192.168.7.5]:28812 at 15:36:21.277729:
03 ----------------------------------------------------------------
04 REGISTER sip:dujinfang.com SIP/2.0
05 Via: SIP/2.0/UDP 192.168.7.5:28812;branch=z9hG4bK-d8754z-...
06 Max-Forwards: 70
07 Contact: <sip:1007@192.168.7.5:28812;rinstance=af02322f2d5c82ba>
08 To: "1007"<sip:1007@dujinfang.com>
```

```
09 From: "1007"<sip:1007@dujinfang.com>;tag=a3e90a4e
10 Call-ID: ZTBhNGVkMTJlOGQzMDZlNjJjN2E2MjQ5NGEzZTZmM2U
11 CSeq: 2 REGISTER
12 Expires: 3600
13 Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE,
SUBSCRIBE, INFO
14 User-Agent: Bria 3 release 3.5.0b stamp 69410
15 Authorization: Digest username="1007",realm="dujinfang.com",
    nonce="40dbf2e3-e5f6-4ee3-badf-855253b12c02",uri="sip:dujinfang.com",
    response="14337381cc0302574b6c39948fbea98f",
    cnonce="0c61ee029710dbf6b740dce279c651fa",
    nc=00000001,qop=auth,algorithm=MD5
16 Content-Length: 0
```

该消息不是第 1 个注册消息，而是在收到 401 响应后，计算出带有 Authorization 头的第 2 个注册消息。其中，我们可以看到，该消息确定发送到了我们在客户端上设置的 SIP Proxy 的 IP 地址上，而与我们以前经常使用的 Domain 中的“dujinfang.com”没有任何关系。在这时，“dujinfang.com”是一个合法的域名，但它没有在 DNS 中解析到 192.168.7.5 这个 IP 地址上。由此你可以看到，在此处的客户端上的 Domain 中，你在练习的时候可以填入任意值，当然填写不属于你自己的域名（如“apple.com”或“microsoft.com”，以及本例子中属于笔者的域名等）是非常不明智的。

第 4 行称为 Request Line，可以看到“dujinfang.com”也出现在这里，同理，它也出现在第 8 行、第 9 行以及第 15 行等。

关于 Domain 我们就先介绍到这里，事实上，如果读者没有亲身的实践，在这里笔者是很难讲清楚的。就连 FreeSWITCH 的作者 Anthony Minessale 也花了一小时在邮件列表中讲这个问题，当然，他讲得是非常有参考意义的。参见 http://lists.freeswitch.org/pipermail/freeswitch-users/2013-August/099131.html。下面我们通过实际的例子来看一下如何通过 Domain 设置多租户，希望读者最终能够理解这里的意思。

15.7.2 配置与实例

上面讲到的注册是可以成功的，而且我们在 SIP 消息中看到到处都是我们设置的 Domain（dujinfang.com）。但是是否注册成功了就说明这支持多租户了呢？答案是否定的。我们还需要做一些工作。首先，我们用以下 Dialplan 测试一下，看 domain 及 domain_name 两个变量能告诉我们什么。

```
<extension name="Domain">
<condition field="destination_number" expression="^1234$">
  <action application="log" data="INFO domain=${domain}"/>
  <action application="log" data="INFO domain_name=${domain_name}"/>
</condition>
</extension>
```

通过上述 Dialplan 的设置，在拨打 1234 后，我们在日志中可以看到如下输出：

```
[INFO] mod_dialplan_xml.c:558 Processing 1007 <1007>->1234 in context default
[INFO] mod_dptools.c:1595 domain=192.168.7.5
```

```
[INFO] mod_dptools.c:1595 domain_name=192.168.7.5
```

可以看出，由于 domain 是一个核心的变量，它继续保持 192.168.7.5 这个 IP 地址值倒也罢了，但 domain_name 也不变，我们根据什么来做多租户的路由呢？即我们根据凭什么说这是某某域里的 1007 发起的呼叫而不是另一个域中的 1007 发起的呢？

其实，这是由于 FreeSWITCH 默认安装要在所有环境下都有运行的根本宗旨决定的，要达到我们的要求，需要检查 Profile（internal）中的如下参数：

```
<param name="force-register-domain" value="$${domain}"/>
<param name="force-subscription-domain" value="$${domain}"/>
<param name="force-register-db-domain" value="$${domain}"/>
```

可以看出，在默认的配置中，上述参数都是生效的，也就是说，不管你的 Domain 怎么配置，都让它强制（force）成了默认的 Domain。因此，在此我们需要删除这些参数（或者在配置文件中注释掉这些行）。然后，重启该 Profile，再次注册时就会看到如下的警告，而且，我们的 SIP 客户端再也注册不上了。

```
[WARNING] sofia_reg.c:2679 Can't find user [1007@dujinfang.com] from 192.168.7.5
You must define a domain called 'dujinfang.com' in your directory
and add a user with the id="1007" attribute
and you must configure your device to use the proper domain
in it's authentication credentials.
```

虽然是个警告，但是个好的警告，它至少告诉我们它在配置文件中找不到这个 Domain 和这个用户了。我们看一看默认的用户目录配置文件 conf/directory/default.xml，它里面定义了一个 Domain，是默认的“$${domain}”值，具体如下：

```
    <include>
<!--the domain or ip (the right hand side of the @ in the addr-->
<domain name="$${domain}">
<params>
    ......
```

找到它后，我们只需要照着它复制一份新的就好了（这里当然也可以把“$${domain}”改成我们想要的“dujinfang.com”，但我们除此之外还要加别的 Domain，因而到后面总是要复制的）。

将上述文件复制到 conf/directory/dujinfang.com.xml，然后修改其中的 domain_name 为如下的值：

```
<domain name="dujinfang.com">
```

在 FreeSWITCH 控制台中执行 reloadxml 使之生效，然后，重新注册客户端，就发现能注册成功了。通过如下的命令也能列出注册信息：

```
freeswitch> sofia_contact 1007@dujinfang.com

sofia/internal/sip:1007@192.168.7.5:34088;rinstance=d3ba18cdf1903f17

freeswitch> sofia status profile internal reg 1007
```

```
Registrations:
=================================================================================
Call-ID:         ZWRhZjA3NjI0NjYxODA1MDdkOGI0YzYxZWVkNGRmZjA
User:            1007@dujinfang.com
Contact:         "1007" <sip:1007@192.168.7.5:34088;rinstance=d3ba18cdf1903f17>
Agent:           Bria 3 release 3.5.0b stamp 69410
Status:          Registered(UDP)(unknown) EXP(2013-10-20 00:56:40) EXPSECS(94)
Host:            seven.local
IP:              192.168.7.5
Port:            34088
Auth-User:       1007
Auth-Realm:      dujinfang.com
MWI-Account:     1007@dujinfang.com
```

接着，拨打 1234 进行测试，就可以看到如下的输出了：

```
[INFO] mod_dialplan_xml.c:558 Processing 1007 <1007>->1234 in context default
[INFO] mod_dptools.c:1595 domain=192.168.7.5
[INFO] mod_dptools.c:1595 domain_name=dujinfang.com
```

从中可以看出，这里的 domain_name 变量也变成我们期望的值了。

15.7.3 进阶

当然，我们新建的 dujinfang.com.xml 与原来的 default.xml 都装入了 conf/directory/default 目录下的用户配置文件。既然是不同的租户，我们就需要把它们分开，因此我们也把整个 default 目录复制一份，放到到新的 dujinfang.com 目录中，并且将 dujinfang.com.xml 中的"include"一行改为：

```
<users>
  <X-PRE-PROCESS cmd="include" data="dujinfang.com/*.xml"/>
</users>
```

还没有完，我们要让自己域中的用户使用单独的 Dialplan 进行路由，因此进入 dujinfang.com 目录，将里面所有的用户配置文件（如 1007.xml）中的 user_context 参数全部改成如下的值：

```
<variable name="user_context" value="dujinfang.com"/>
```

执行 reloadxml 命令后再拨打 1234 发现打不通了，原因很显然，我们还没有在 Dialplan 中设置 dujinfang.com 这个 Context。当然，设置它也很简单。进入 conf/dialplan/ 目录，将 default.xml 复制为 dujinfang.com.xml，并将里面的 Context 的名称修改为我们需要的值，如：

```
<include>
<context name="dujinfang.com">
```

再一次执行 reloadxml 命令并拨打测试，就可以看到，我们这里的路由也跟默认的配置完全分离了。这就基本上达到了我们想要的效果。读者接下来可以参照这里的步骤再配置几个新的域以对比测试。

15.7.4 其他

使用多 Domain（及多 Directory、多 Dialplan）支持多租户以后，有以下几个事情是需要注意的：

- Dialplan 中的呼叫字符串不能再使用类似“user/1000”这样的缩写形式了，而要写成如“user/1000@${domain_name}”的形式。
- 在使用会议、fifo 等类的应用时，注意不要像本书中的例子一样图省事[⊖]把会议的名称写成“3000”或把 fifo 的名称写成“book”，一定要写成如“3000-${domain_name}”或“book@${domain_name}”之类带 Domain 的长名字。
- 在实际应用中 Domain 可以与实际的 FQDN 相同，最好能使用 DNS 的 SRV 记录进行解析。不同的机构可以使用不同的域，如 dujinfang.com、microsoft.com 等；同一个机构不同的分支机构间也可以使用不同的域，如 beijing.dujinfang.com、shanghai.dujinfang.com 等。
- 如果需要计费，话单等也是需要特别注意的。
- 除此之外，还可以进一步细分，将不同的租户分到不同的 Sofia Profile 中去。即每个租户都有自己的的 Profile（分别占用不同的端口号），它们的 Profile 中的信息及各种参数都可以不同。当然，若所有的租户都想使用 5060 这样的端口号，则可以使用 DNS 的 SRV 记录来解决。关于 SRV 记录的用法超出了本书的范围，读者可以自行研究。

15.8 使用 loopback Endpoint 外呼

如果要在 FreeSWITCH 中向外发起呼叫，需要一个呼叫字符串。在前面的章节中，我们学习过几种呼叫字符串，如 user/1000、sofia/gateway/gw1/xx、ftdm/1/a/1000 等。

但有些情况下，找到呼叫字符串不是那么容易。比方说，我们做了一套企业应用，该企业比较大，有很多分支机构，而且由于历史原因号码分配比较乱，因此我们需要在 Dialplan 中设置了几百个 Extension，以匹配不同号码段正确路由，比如下面的代码：

```
<condition field="destination_number" expression="^0(.*)$">
<condition field="destination_number" expression="^11(2.*)$">
<condition field="destination_number" expression="^1234(852.*)$">
<condition field="destination_number" expression="^1234(52.*)$">
<condition field="destination_number" expression="^1234(.*)$">
......
```

如果在我们的应用中，在某些场合下（如自动传真、催缴费等）需要在 FreeSWITCH 中主动呼叫这些号码，则我们可以使用 originate 进行外呼，但在外呼时需根据不同的号段我们生成不同的呼叫字符串。这样我们的程序中就需要另外一套路由数据，这不仅与 Dialplan 中的数据重复，而且不利于维护。因此，我们就考虑能否在用 originate 外呼的时候也使用

⊖ 当然，也不完全是为了省事，有时候是因为太长的行在书中不容易排版。

Dialplan 呢？答案是肯定的，这里我们使用一个新的 Endpoint，称为 loopback。在理解它的实用原理之前，我们先来看一个简单的例子。

我们知道，路由到 9664 后会播放保持音乐，而路由到 9196 后会执行 echo。以前只有有了一个 Channel 以后才能进入 Dialplan，并路由到 9664，而现在我们可以使用 loopback 提前进入 Dialplan 进行路由。如下面的命令：

```
freeswitch> originate loopback/9664 &bridge(loopback/9196)
```

这些 Channel 都是在 FreeSWITCH 内部发生的，运行完毕后可以使用下列命令看到，上述命令建立了 4 个 Channel（由于输出行太长，我们简单了输出，仅列出有代表性的字段）：

```
freeswitch> show channels

22937c99-7710-4f64-869f-4f441cc15b54,inbound,loopback/9664-b ...
66f83997-5115-43e9-b1b1-b786ac64a9d2,outbound,loopback/9664-a ...
d91f7824-b314-4996-acc4-e27fb846e66a,outbound,loopback/9196-a ...
57416b98-702b-4d38-bc1b-084966fd04e3,inbound,loopback/9196-b ...
```

那么为什么它有 4 个 Channel 呢？我们知道在普通情况下，如果执行如下呼叫，将只产生两个 Channel：

```
freeswitch> originate user/1000 &bridge(user/1001)
```

loopback 这个 Endpoint 的关键就在这里，当然它的实现原理很简单但讲起来却很复杂。为了能让让呼叫先到达 Dialplan 查找路由，它首先创建一个假的腿（即 Channel，称为 loopback-a，我们称为第一条腿），然后让该腿进入 Dialplan 进行路由，执行 playback 播放保持音乐，然后再创建另一条腿（称为 loopback-b，我们称为第二条腿），以继续下一步的动作。在此，它执行 bridge。我们在 bridge 中也使用了 loopback，因而会发生同样的事情，先创建一条假腿（另一个 loopback-a，第三条腿）到 Dialplan 中找到适当的路由，执行 echo，然后又创建一条腿（第四条腿）再与原先的那个腿 bridge 起来。如果还是不理解的话，可以查看一下刚才呼叫的日志，就能比较清楚了：

```
[NOTICE] switch_channel.c:1044 New Channel loopback/9664-a [66f83997-...
[NOTICE] switch_channel.c:1042 Rename Channel loopback/9664-a->loopback/9664-a
[NOTICE] switch_channel.c:1044 New Channel loopback/9664-b [22937c99-...
[NOTICE] mod_loopback.c:946 Channel [loopback/9664-a] has been answered
[NOTICE] mod_dptools.c:1225 Channel [loopback/9664-b] has been answered

[NOTICE] switch_channel.c:1044 New Channel loopback/9196-a [d91f7824-...
[NOTICE] switch_channel.c:1042 Rename Channel loopback/9196-a->loopback/9196-a
[NOTICE] switch_channel.c:1044 New Channel loopback/9196-b [57416b98-...
[NOTICE] mod_loopback.c:946 Channel [loopback/9196-a] has been answered
[NOTICE] mod_dptools.c:1225 Channel [loopback/9196-b] has been answered
```

从日志中可以看到，一共有 4 条腿（4 个 Channel）参与到该通话中，它们的名字分别是 loopback/9664-a、loopback/9664-b、loopback/9196-a、loopback/9196-b。

当然，你也可以执行以下的例子，它将先呼叫 1000，再呼叫 1002，仍然会产生 4 条腿：

```
freeswitch> originate loopback/1000 &bridge(loopback/1001)
```

下面的命令的效果是一样的，但只会产生 3 条腿。

```
freeswitch> originate loopback/1000 1001
```

这是因为，1001 那一端没有使用 loopback，在 1001 那一端产生了两条腿以后，呼叫进入 Dialplan 进行路由，然后又“bridge”到 1001，产生了第三条腿。

理解了这些之后，就可以对本节最开始的时候提出的那些号码进行呼叫了，而不用管它们是具体怎么路由的，如呼叫某个号码并自动发送传真：

```
freeswitch> originate loopback/123485234xxx &txfax(/tmp/test.tiff)
```

需要注意的是，由于使用 loopback 将比普通的呼叫多产生一条腿，因而话单可能受影响。为此，所有的 loopback 的 Channel 中都设置了一个 is_loopback 变量，以方便其他程序（如计费等）进行后续处理。

为了最大限度地减少因额外的腿产生的影响，也可以通过一个通道变量让这条额外的腿在完成它的使命之后尽快释放（或者说两条腿合并为一条），如下：

```
freeswitch originate {loopback_bowout=true}loopback/1000 &echo
```

上述命令将先建立两条腿，呼叫 1000，在成功执行 echo 命令时，两条腿将条合并为一条。读者可以使用 show Channels 命令进行查看。

本节我们花了很大篇幅给大家讲解了 loopback，之所以这么做除了因为它在某些场合确实有用外，还因为我们为了给大家讲清楚默认 Dialplan 中的 Local_Extension 部分的路由。在 6.1.9 节中，我们为了方便讲解，把默认 Dialplan 中呼叫失败后自动入进入 Voicemail 的 Dialplan Action 改写成了如下的形式：

```
<action application="voicemail" data="default ${domain_name} ${dialed_extension}"/>
```

但实际上，它是如下的形式：

```
<action application="bridge" data="loopback/app=voicemail:default ${domain_
name} ${dialed_extension}"/>
```

后面这种方式使用了 loopback。在大多数情况下，两者的效果是一样的。只不过，后者因为能在进入 Voicemail 后能产生一个 Channel，因而还可以配合 att-xfer（见第 15.1.2 节）一起使用。

在这里，与前面讲的 loopback 会先进入 Dialplan 进行路由不同，它直接使用“app=”这样的语法执行一个 App，相当于 loopback Dialplan 的 inline 形式（关于 inline Dialplan 参见 6.2 节）。

15.9　在 Web 浏览器中打电话

多年来，随着互联网的发展，能在网页上打电话已经成为人们的一个梦想。当然，这些年来，人们也实现了一些方案，这些方案一般都是基于浏览器的插件实现的，如基于 IE 浏览器的 ActiveX 插件、Java Applect 插件以及 Flash 插件等。其中，基于 Flash 插件的产

品和解决方案由于其跨平台性和兼容性比较好，使用比较广泛，如比较著名的 Red5 Media Server[1]及 Wowza Media Server[2]等。

然而，随着 HTML5 技术的发展以及几年前苹果公司声明在 iOS 设备中不再支持 Flash，Flash 的使用量开始下滑，并且普遍认为前景不好。而此时，谷歌提出的基于 HTML5 的 WebRTC 实时通信技术却受到了大家的关注，各在线通信平台也纷纷推出支持 WebRTC 的实时通信方案。

但无论如何，现在的 IE 浏览器不支持 WebRTC 还是铁定的事实，所以我们还是需要 Flash。下面我们就来看一下 FreeSWITCH 对 Flash 和 WebRTC 的支持是如何实现的。

15.9.1 Flash

基于 Flash 的实时多媒体通信是基于 Adobe 的 RTMP 协议进行的。FreeSWITCH 中通过 mod_rtmp 实现了一个基于 RTMP 协议的 Endpoint，可以支持用 Flash 实现的软电话。如我们经常讲的一样，虽然 Flash 前景不再被看好，但是在一定范围内它还将顽强地存在。而且，作为有别于 SIP 模块（“mod_sofia”）的另外一个 Endpoint，也很有参考和借鉴意义。

在 FreeSWITCH 源代码目录中使用如下命令即可安装该模块：

```
# make mod_rtmp-install
```

在 FreeSWITCH 控制台上使用 load mod_rtmp 命令加载该模块后，它将监听 RTMP 协议默认的 1935 端口，并等待客户端连接，使用如下命令将可以显示该模块的有关状态：

```
freeswitch> rtmp status

default tcp:0.0.0.0:1935     profile
```

上面的命令显示了有一个 RTMP 的 Profile 运行在 1935 端口上。

FreeSWITCH 源代码中也实现了一个 Flash 版的软电话，并提供了客户端的例子。客户端及例子页面的源代码在 FreeSWITCH 源代码目录的 clients/flex/ 目录中。要运行该例子，我们首先要进行一些修改，将 freeswitch.html 中的 rtmp_url 参数改为指向我们自己的 RTMP 服务器。如原来的参数值如下：

```
rtmp_url: 'rtmp://my.ip.address.here/phone'
```

笔者在测试时修改后的值如下（它指向笔者本机上的 RTMP 服务，即我们上面打开的 1935 端口上的服务）：

```
rtmp_url: 'rtmp://127.0.0.1/phone'
```

当然，为了能访问该 HTML 页面，我们还需要把这些相关的 Web 资源都放到一个 Web 服务器上运行。Python 语言提供了一个简单的方案用于运行一个简单的 Web 服务器，我们可以在客户端的源代码目录中使用下载命令启动一个简单的 HTTP Server，它将默认运行在

---

[1] 参见 http://www.red5.org/。

[2] 参见 http://www.wowza.com/。

8000 端口上。

```
# python -m SimpleHTTPServer
```

然后，打开浏览器，访问与该服务器对应的网址，如 http://localhost:8000/freeswitch.html，就可以看到图 15-8 所示的界面了。

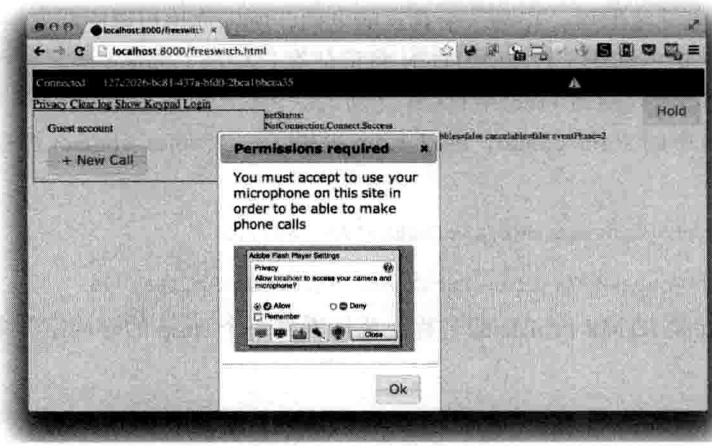


图 15-8 Flash 测试页面初始界面

其中，可以看到在最左上角有 Connected 字样，表示我们已经跟 RTMP 服务器（即 FreeSWITCH）连接上了。但是，由于该网页上的 Flash 软电话需要访问本地的麦克风，由于网页的安全性考虑，系统弹出一个窗口以提示用户是否允许访问这些设备。这里我们选择允许（Allow）就可以了。

接下来，单击 Login 链接，输入用户及密码进行登录（即注册），笔者的输入如图 15-9 所示。

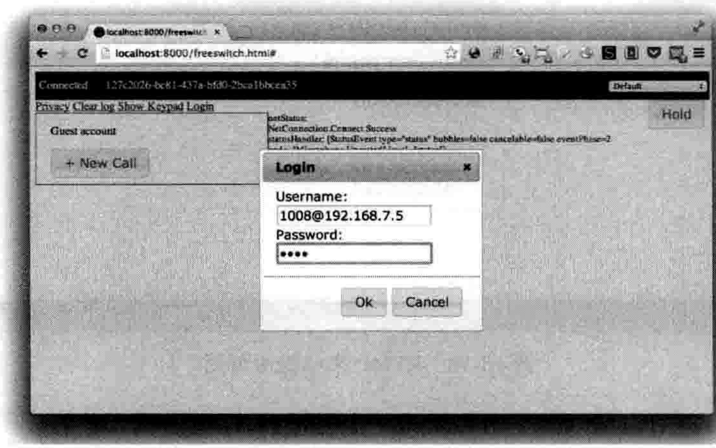


图 15-9 将 Flash 软电话登录到 FreeSWITCH 上

其中，“1008@192.168.7.5”为默认用户目录的配置，相当于“ user@domain”，密码也是默认的“1234”。即“ mod_rtmp”也跟“ mod_sofia”一样使用用户目录中的配置对用户进行验证。

注册成功后，我们就可以单击“New Call”按钮输入一个号码拨打电话了，作为第一个测试我们可以拨打 9196。

当然，Flash 电话注册后，它是一个真正的电话，所以你也可以呼叫它。

回忆一下 SIP 中的注册，FreeSWITCH 将记住客户端的 Contact 地址。在 RTMP 中也类似，每当有 RTMP 客户端向 FreeSWITCH 注册时，FreeSWITCH 也记录该客户端的地址，如类似 mod_sofia 中的 sofia_contat，mod_rtmp 提供了一个 rtmp_contact 命令可以查询客户端的注册情况：

```
freeswitch rtmp_contact default/1008@192.168.7.5

rtmp/127c2026-bc81-437a-bfd0-2bca1bbcea35/1008@192.168.7.5
```

其中，default 是 RTMP Profile 的名称。我们看到一个 rtmp 的呼叫字符串，然后就可以尝试呼叫它了：

```
freeswitch> originate rtmp/127c2026-bc81-437a-bfd0-2bca1bbcea35/1008@192.168.7.5
&echo
```

执行上述命令后，就会在浏览器中听到来话的提示音，并且可以看到如图 15-10 的来电提示页面。

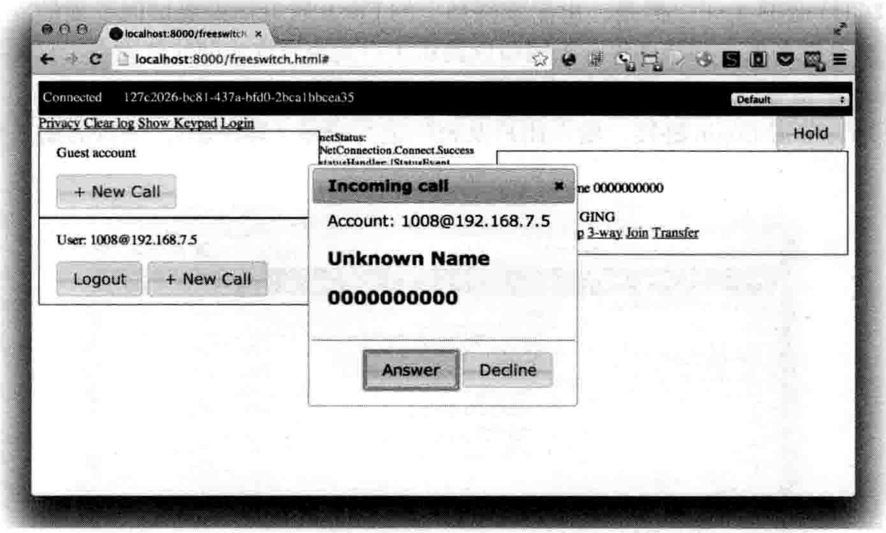


图 15-10 RTMP 来电提示页面

测试成功后，就可以编辑如下的 Dialplan 让别的 Flash 电话或 SIP 电话呼叫到 1008 了：

```
<extension name="RTMP">
```

```
    <condition field="destination_number" expression="^(1008)$">
      <action application="bridge"
              data="${rtmp_contact(default/$1@$${domain})}"/>
    </condition>
  </extension>
```

该 Flash 电话客户端提供源代码，也提供 JavaScript API 用于在 HTML 网页中调用，因而可以很方便地集成到其他 Web 系统中去。

15.9.2 WebRTC

WebRTC[1]的全称是 Web RealTime Communication，即基于 Web 的实时通信。实际上 WebRTC 提供了在浏览器中使用 JavaScript API 访问本地的声音和视频设备的手段。由于保护隐私和安全性的原因，浏览器在每次使用用户的声音或视频设备时都会询问是否允许。

注意

目前 Chrome、FireFox 以及 Opera 等浏览器实现了 WebRTC。

值得注意的是，WebRTC 仅提供了访问音视频设备这些媒体资源的方法，并没有规定信令是怎么走的。现有的例子大部分都使用了 Google App Engine Channel API 传输信息控制协议。另外一个可以传输信令的就是 WebSocket[2]。

WebSocket 是 HTML5 最初提供的一种浏览器与服务器间进行双向全双工通讯的网络技术。大家熟知的 HTTP 协议仅能用于通过浏览器单向请求服务器资源，而为了能使服务器主动向浏览器客户端推送资源，人们想尽了办法，如使用 Ajax 轮循、HTTP 长连接、Flash 提供的 Socket 功能及 IE 中的 ActiveX 控件等。WebSocket 的出现解决了这些连接的混乱，它可以直接使用浏览器提供的功能和 API 与后端建立双向的 Socket 连接，包括 IE9 在内的现代浏览器都支持 WebSocket。

有了 WebSocket 以后，SIP 的承载方式就又多了一个选择。大家都已经知道，SIP 一般通过 UDP 承载，也可以通过 TCP 或 TLS 承载，而出现了 WebSocket 以后，它也可以承载 SIP，这种技术就称为 SIP over WebSocket[3]，它目前还是一个 IETF 的草案。

信令和媒体都有了，我们就可以进行通信了。需要注意的是，浏览器中的视频编码有两个阵营。一个是以 Google Chrome 为代表的，它支持的视频格式是 VP8；另一个是以 Apple Safari 为代表的，支持 H264，而 FireFox 也支持 H264。由于 FreeSWITCH 不支持转码，目前这两种视频还是不能互通。

另外，在写作本书时，JsSIP 对 FireFox 的支持也不是很好，因而在以下的例子中我们都使用 Chrome 浏览器。

---

[1] http://www.webrtc.org/。

[2] 参见 http://zh.wikipedia.org/wiki/WebSocket http://tools.ietf.org/html/rfc6455。

[3] 注意，不要把 WebSocket 跟 WebRTC 混了，它们是完全不同的东西。参见 http://datatracker.ietf.org/doc/draft-ietf-sipcore-sip-websocket/ 。

1. JsSIP

JsSIP [1]是一个开源的社区项目。FreeSWITCH 官方提供了基于 JsSIP 的 WebRTC 的例子，地址为 https://webrtc.freeswitch.org/。

读者访问以上地址可以看到网页提示输入一个名字或分机号进行登录，实际上就是向 FreeSWITCH 注册。FreeSWITCH 后台开启了 accept-blind-auth，因而任何用户名均可以成功注册。

登录成功后就可以在“To”一栏输入号码进行呼叫了。如输入 888 就可以进行 FreeSWITCH 默认的会议系统；输入 mcu 可以进入一个由第三方 MCU 提供的视频会议系统等。

2. sipML5

sipML5 [2]是由 Doubango 团队做出的 SIP Over Websocket 的开源实现。该项目可以在线试用，也可以在自己本地试用。sipML5 的源代码是他们 SVN 维护的，使用以下命令可以将该项目检（checkout）到本地：

```
svn checkout http://sipml5.googlecode.com/svn/trunk/ sipml5-read-only
```

如果你使用 Apache 或 Nginx Web 服务器，可以把这些代码放到你的 Web 服务器目录下，然后在浏览器上访问。如果你的机器上有 Python 的话，可以使用它的 SimpleHTTPServer 模块很简单地启动一个 Web 服务器，如以下命令将启动一个简单的 Web 服务器[3]，默认使用 8000 端口：

```
cd sipml5-read-only
python -m SimpleHTTPServer
```

如果你想改变端口，如改到 8888，则可以使用如下命令：

```
python -m SimpleHTTPServer 8888
```

打开你的浏览器，访问该服务器对应的端口，就能看到与 sipml5.org 同样的界面[4]。然后，单击“Enjoy our live demo”按钮就可以看到 WebRTC 电话的页面了。当然也可以尝试直接访问下面这个地址：

```
http://localhost:8000/call.htm
```

由于你是在自己的服务器上使用的，因此要先进入“Expert mode”，将“WebSocket Server URL”修改为你服务器 WebSocket 的监听地址，修改完毕后单击“Save”按钮保存。这些值将保存到本地浏览器的 Local Storage 里，因而也可以直接通过浏览器的调试模式修改这些值，如图 15-11 所示。

---

[1] 参见 http://jssip.net/。

[2] 参见 sipml5.org/。

[3] 本例是使用 Python2 的例子，如果使用的是 Python3，则可以使用下列命令：“python3 -m http.server”。

[4] 不过，作者在测试中发现，由于 index.html 页面引用了一个外部的 JavaScript，而该地址在国内是打不开的，因而会导致打开这个页面很慢。要解决这个问题，可以在 index.html 中找到包含下面地址的行并把它删掉：https://apis.google.com/js/plusone.js。

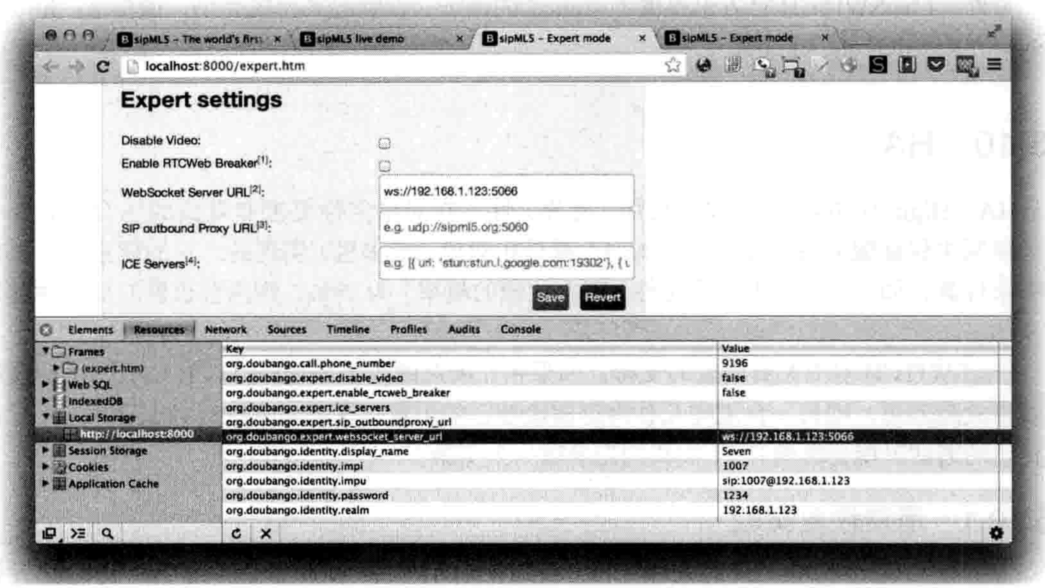


图 15-11　在 Expert 模式进行设置

保存完上述设置后，即可以返回原先的 call.htm 页面。输入相关信息就可以注册（Login）了。图 15-12 显示了笔者填写的注册信息，并演示了注册后测试呼叫 9196 的情况。

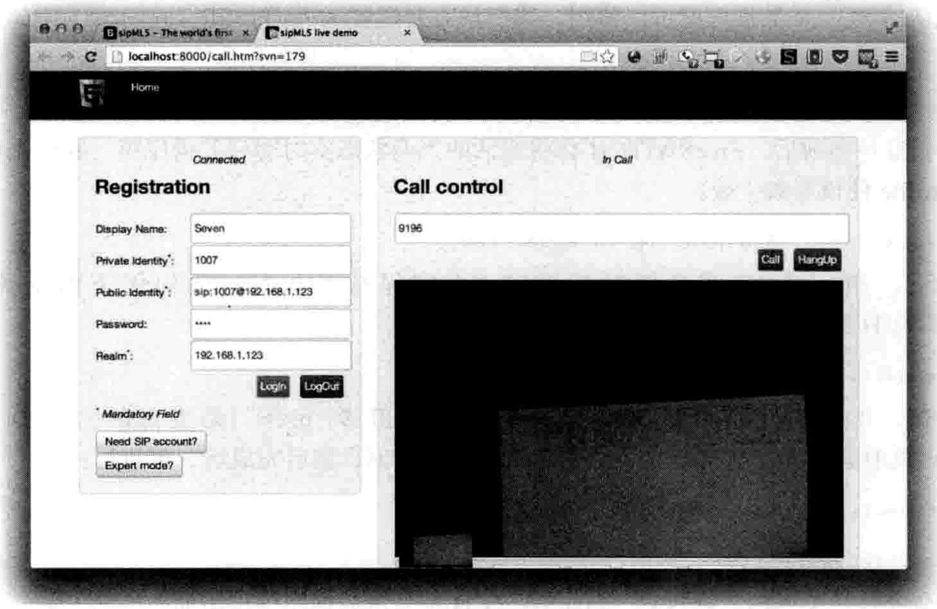


图 15-12　使用 sipML5 进行 WebRTC 视频呼叫

另外，FreeSWITCH 官方也提供了 sipml5 的例子，设置起来比较简单。请参考：https://webrtc.freeswitch.org/sipml5/。

15.10 HA

HA（High Availability）即高可用（可靠）性。在对稳定性要求非常高的场合，需要一定的机制来保证服务的稳定。一般的 HA 都是用双机（或多机）实现的，它的理论是，根据概率来计算，如果一台机器的稳定性（不出故障的概率）为 99%，则两台机器的稳定性就是 99.99%。

FreeSWITCH 对 HA 有内置的支持，如果相互配对的两个 FreeSWITCH 中有一个崩溃或因硬件故障宕机，则另一台能在几秒内迅速接替它的工作，保证正在进行的通话不断话（正在接续的通话可能受影响）。下面我们先来做一个实验。

15.10.1 崩溃恢复实验

为了使用实验简单，我们先用一个 FreeSWITCH 实例来做实验。首先，我们配置好一个 FreeSWITCH 实例，用一个 SIP 客户端发起一路通话呼叫 9664 听保持音乐；然后让 FreeSWITCH 崩溃，这时候客户端会短暂地听不到声音；接着，我们重启 FreeSWITCH 并恢复崩溃前的通话，客户端上就又重新听到的声音。

为了让 FreeSWITCH 能在崩溃后恢复，FreeSWITCH 需要将当前通话的信息写到数据库中。默认的 SQLite 数据库就可以使用，我们只需要配置 Sofia Profile（internal）中如下一个参数：

```
<param name="track-calls" value="true"/>
```

配置好该参数后，FreeSWITCH 会将通话的当前状态实时写到数据库里，我们先重启一下该 Profile 使该参数生效：

```
freeswitch> sofia profile internal restart
```

然后，用一个 SIP 客户端呼叫 9664，听到保持音乐后，我们用下面的命令让 FreeSWITCH 崩溃：

```
freeswitch> fsctl crash
```

当然，上述命令是一个测试的命令，与“kill -9 <pid>”是类似的。总之，至此 FreeSWITCH 就崩溃了。然后，我们重启 FreeSWITCH，重启完成后，再执行：

```
freeswitch> sofia recover

Recovered 1 call(s)
```

该命令是最关键的。通过它，FreeSWITCH 就会从数据库中读出崩溃前的通话信息，并恢复。

当然，该实验能成功的前提是你做得足够快（通过几秒或几十秒），因为有些 SIP 客户

端在长时间收不到 RTP 流后，就会自己挂机，客户端挂机了，当然在 FreeSWITCH 端再恢复也没什么用了。

其实这里能恢复的原理很简单。由于在 SIP 会话建立后，电话进入正常通话中，一般在挂机之前再没有 SIP 消息交互了，只有 RTP 用于传送媒体数据。如果 FreeSWITCH 崩溃，则 RTP 数据只是短暂不能收发，而由于 RTP 是 UDP 的，因此客户端感觉不到 FreeSWITCH 崩溃而仍然发送 RTP 流（也许有一点点感觉就是暂时收不到 RTP 流了，而短时间内收不到 RTP 流的话客户端也不会抱怨）。因而在 FreeSWITCH 恢复后，电话马上就恢复了。当然，FreeSWITCH 在恢复通话后，会重新发一个 re-INVITE 消息，以确定对方是否还在。

注意

我们这里讨论的崩溃恢复是假设 SIP 采用 UDP 的情况。如果 SIP 采用 TCP 方式连接，由于 TCP 是面向连接的协议，就不能保证总是能够恢复成功（跟对端和网络环境有关）。

15.10.2 HA 简介

上一节我们讲了 FreeSWITCH 中崩溃恢复的原理和一些细节，严格上来说还不算 HA，但它为 HA 的实现打下了基础。其实，HA 一般是为了解决硬件问题的。因为，如果软件有问题，如 FreeSWITCH 崩溃后，可以用另外一个监控程序迅速重启并恢复通话。但如果硬件发生问题，上述方式就不能解决了，因而需要配置另外一台机器，在检测到其中一个失败后便立即接替服务。

为了能使一台主机能完全接替另一台主机提供同样的服务，两台主机必须完全相同。下面我们以两台 FreeSWITCH 主机为例来说明一下。

如图 15-13 所示，有两个 FreeSWITCH 节点，分别为节点 1 和节点 2，它们分别具有 IP 地址 10.0.0.11 和 10.0.0.12。其中，节点 1 为主用节点，因此它绑定了一个浮动 IP 10.0.0.10。对于 SIP 终端来讲，它们不知道节点 1 和节点 2 的存在，只知道它在跟一个 IP 地址是 10.0.0.10 的 SIP 服务器进行通信。现在，IP 地址 10.0.0.10 绑定在节点 1 上，如果任何时候节点 2 发现节点 1 故障，它就会将 IP 地址 10.0.0.10 绑定到自己的网卡上，并接替节

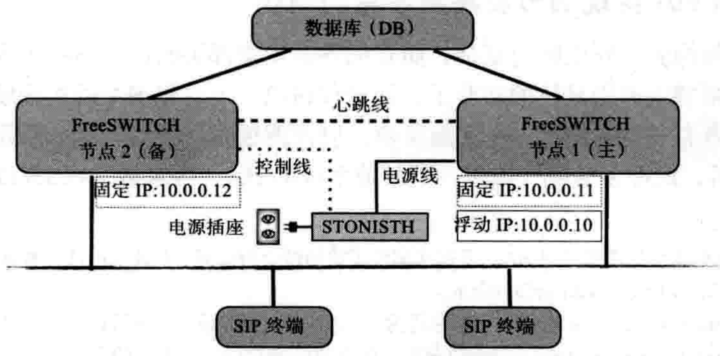


图 15-13 FreeSWITCH HA 拓扑结构示意图

点 1 继续进行工作。

为了能使节点 2 能实时监测到节点 1 的健康状态，两者之间使用心跳线连接，并定期交换心跳（Heartbeat）消息。如果节点 2 在规定的时间内（通常几秒内）收不到对方的心跳，则认为节点 1 故障，并触发倒换过程，自己变为主节点。在物理上，心跳线可以用串口线，也可以用以太网线。

如果节点 1 在一段时间内恢复，这时候一般有两种选择：①它自动变为备用节点，下次节点 2 再出故障时它再变为主节点；②它依然作为主节点，节点 2 则变回从节点。具体的倒换策略视具体应用情况而定。

为了能使节点 2 能恢复节点 1 上正在进行的通话，两个节点需要连接到一个共享的数据库[1]。

可以想象，节点 2 对节点 1 健康状况的判断仅依靠心跳线，如果心跳线出现故障，则可能引起错误的判断，造成不必要的主备切换。更严重的是，网络上会出现两个主机具有相同的 IP 地址（即浮动 IP 为 10.0.0.10），两台主机也可能同时访问数据库中的共享资源引起数据混乱。这种可能的错误称为脑裂[2]。为了预防脑裂的发生，一般有以下两种解决办法：

- 再增加一条心跳线，以降低由于心跳故障引起的错判；
- 增加 STONITH [3]设备，强制关闭另一台设备的电源。

当然，到这里读者也可以看到，如果节点 1 真的发生故障了，那么节点 2 检测到以后便会启动倒换过程。如使用 STONITH 工具关掉对方节点的电源、将浮动 IP 绑定到本机上、启动 FreeSWITCH、恢复通话等。这些工作可以使用一些脚本来解决，也可以使用专用的 CRM [4]软件解决。FreeSWITCH 可以很好地配合各种 CRM 软件工作，不过，在本书中，限于篇幅，我们就不涉及了。

需要指出，本节讲的是通用的 HA 的基本知识，我们以 FreeSWITCH 作为例子进行讲解只是为了能方便直观一些，其他的系统 HA 也与之类似，如在数据库 HA 系统中，可以把这里的 FreeSWITCH 节点看做是数据库节点，而这里的共享数据库资源就可以看作是数据库 HA 系统中的共享存储（磁盘阵列、SAN 存储等）。

15.10.3 双机 HA 实现细节及需要注意的问题

由于搭建实际的演示环境颇为复杂，而且把各项步骤都讲明白也需要很大的篇幅，因此笔者就不在这里带领大家做具体的实验了，而是仅讨论一下双机 HA 的实现细节及需要注意的问题。如果读者有其他系统的 HA 实施经验，应该能很容易的实现；如果没有其他系统的 HA 实施经验的话，最好多学习和参考一下其他系统的 HA，再来做 FreeSWITCH 的实验。

---

[1] 当然，数据库也可能需要实现 HA，不过那超出了我们讨论的范围，在此我们认为数据库是可靠的。

[2] 参见 http://linux-ha.org/wiki/Split_Brain。

[3] Shout the Other Node In the Head，可以翻译为"拍死它"。这种设备一般可以通过串口进行控制。

[4] CRM 是 Cluster Resource Management 的缩写，即资源管理软件，与常见的客户关系管理系统（Customer Relationship Management）完全不同。

首先，我们需要两台一模一样的主机，安装操作系统（我们仅讨论 Linux）和必要的软件，并安装 FreeSWITCH。然后安装心跳软件。一般来说，可选的心跳软件有 Keepalived 和 Heartbeat，前者比较古老，后者比较新，它们都能在检测到心跳失败时调用相关的脚本切换 IP 和 FreeSWITCH。

当发生主备机切换时，需要在备机上启动 FreeSWITCH，然后执行 sofia recover 命令，以恢复话务。这些过程都可以写一个简单的 Shell 脚本来实现，也可以直接配置 FreeSWITCH（在 vars.xml 中）在系统启动时自动执行 sofia recover，如：

```
<X-PRE-PROCESS cmd="set" data="api_on_startup=sofia recover"/>
```

当然，FreeSWITCH 的过程可能会比较慢，因而会延长总体切换的时间。为了节省启动时间，我们可以使用 FreeSWITCH 的 StandBy 模式，即先在备机上启动 FreeSWITCH，让其什么也不干，等到 IP 切换过来后就可以立即执行 sofia recover 以恢复业务。为了能支持这种模式，需要修改操作系统内核参数，允许 FreeSWITCH 监听在“不存在”的 IP 地址上（因为以后要使用的浮动 IP 现在还绑定在主用节点上，备用节点上还没绑定）。在 Linux 上可以使用如下命令实现：

```
echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind
```

上述命令会动态更改内核参数，但如果想使该参数在系统重启后也生效，需要将配置选项写到系统配置文件中，如：

```
echo “net.ipv4.ip_nonlocal_bind=1” >> /etc/sysctl.conf
```

当然，为了能正常恢复话务，还需要在 FreeSWITCH 的 Sofia Profile 中配置 ODBC（或 PGSQL）数据库支持以及开启 track_calls 参数。除此之外，还要注意配置文件中 domain 的值，注意这需配一个公共的值（如使用浮动 IP 或一个域名）而不是使用默认的、本机的固定 IP 地址。

其实 FreeSWITCH 的 HA 没有什么神秘的，相信如果读者有一些 Linux 经验并参照本节中所讲的知识应该都能配置出来。只是在实际应用中更需要注意以下几个问题：

- **性能降低**。为了能实现 HA，需要把通话数据实时写到数据库中，这无形地就增加了资源占用，降低了系统的吞吐量。
- **成本增加**。增加的另一台配对主机以及需要增加的网络设备至少需要 2 倍的硬件成本，也需要增加维护成本，并且收效不是那么明显——如果一台永远不出故障，钱就白花了；而且，如我们在 15.10 节讲的，它提高的可靠性却有限（从 99% 增加到 99.99% 只是增加了 0.99% 的可靠性[⊖]，显然与投入的成本不成正比）。
- **人为原因**。并不是配置好 HA 就能放心睡大觉了，一个成熟的系统需要大量的测试以及长时间的优化；另外，HA 也并不是解决一切问题的良药，更多的系统故障都是人为原因造成的。
- **带来的好处**。当然，它带来的好处就是稳定性确实提高了。其实，更常用的功能就是

⊖ 当然，这个问题分怎么看。其实故障率是从 1% 下降到 0.01%，相当于故障率下降了 100 倍。

系统升级时可以先升级备用的主机，然后进行主备倒换，再升级另一台主机而不会影响业务。

- 其他。FreeSWITCH 在底层实现了从崩溃中快速恢复通话的机制，剩下的就要靠在使用过程中应用层的发挥了。最典型的，在大规模的集群中使用 N+M [⊖]的策略应该是对投入和产出最好的平衡。

15.11 集群及分布式部署

在上一节我们讨论了 HA，与 HA 相对的另一个概念是 HP（High Performance），即高性能。它原先主要指大规模的并行计算，现在延伸到互联网域，就是大规模的集群。大规模的集群也是当下一个更时髦的词——“云”——的基础。由于这种方式一般采用多节点负载分担的方式工作，因而又称为负载均衡（Load Balance，LB）。

15.11.1 大规模集群的总体结构

图 15-14 显示了一个基于互联网的大规模的集群的拓扑结构。一般来说，用户通过客户端（Client）访问网络上的服务。客户端会使用 DNS 查询找到它要访问的主机。DNS 服务一般使用轮循的方法，通过返回多个 IP 地址，将客户端的访问流量分散到不同的 IP 地址上。另外，有的 DNS 服务也可以根据用户的来源（地理位置）返回不同的 IP，以将用户分配到就近的数据中心的服务器上。

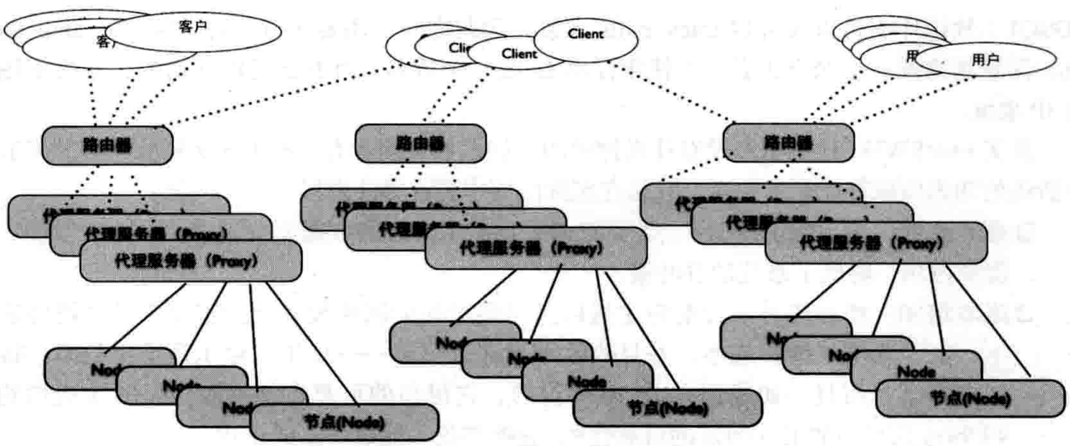


图 15-14 大规模集群总体结构示意图

当客户的请求到达某一数据中心后，前端的路由器设备（或以太网交换机）会根据访问的服务进行合理的分发，将请求发到后端的代理服务器（Proxy）上。这些路由器比较有代

⊖ 即 N 台主机运行业务，M 台备用，我们本节讨论的 HA 其实是 N=M=1 的特殊情况。

表性的有硬件的 F5 或软件的 LVS [1]等，它们都是在 IP 层（或应用层）进行转发分配的。

当请求到达某一个代理服务器后，它会检查本地的设置，将请求数据再转发到某一个计算节点（Node）上。而后端的计算节点间也可以通过内部的网络进行通信，或访问共享的数据等。最后，计算节点将执行相关的服务程序对收到的请求进行服务，有必要的话，它也可能查询数据库或与其他系统交互。

总之，互联网上海量用户的请求就是这样通过前端的 DNS、路由器以及代理服务器分配到不同的计算节点上，并由其对用户进行服务的。

具体来讲，对于支撑整个互联网基础的 Web 应用，比较典型的代理服务器有 HA-Proxy、Nginx、Apache 等；后端的计算节点可以认为是 Apache/PHP、Tomcat（Java）、Thin/Mongrel/Unicorn（Ruby/Rails）、Python 等（当然，实际的情况还涉及后面的数据库、消息队列、大型存储等，比这里讲的要复杂得多）；而对于我们关心的 VoIP 应用来说，常用的代理服务器有 OpenSIPS 和 Kamailio 等，后端的节点可能是 FreeSWITCH 或 Asterisk。当然，VoIP 通信与 Web 服务不同的是，VoIP 对于实时性的要求也比较高，而且信令跟媒体也是可以分离的（因而也有专门代理媒体的代理服务器，如 RTPProxy）。

在实际的应用中，这些代理服务器应该足够“聪明”，即首先它能够比较智能地分配客户端的请求，力求使后端的计算节点负载比较均衡；其次，如果后端的计算节点出现故障，它也能很快探测到，并停止向其分发客户端的请求[2]。当然，为了最大程度地保持稳定，代理服务器甚至前端的路由器及网交换机等都可以进行成对配置。

15.11.2　负载均衡配置实例

上一节，我们讲了大规模的集群的总体结构。具体到 FreeSWITCH，它最典型的应用就是作为集群中的一个节点提供具体的服务，而前面的 DNS、路由器以及各种代理服务等则主要是负责将相应的注册和呼叫等请求分配到相应的 FreeSWITCH 服务器上进行处理。

下面我们来做一个负载均衡的实验。如图 15-15 所示，以一台 OpenSIPS 和两个 FreeSWITCH（A、B）节点为例[3]。OpenSIPS 位于前端，做 Proxy；FreeSWITCH 在后端，进行话务处理；它们都共享同一个数据库。OpenSIPS 用于分发 SIP 请求，而后面的 FreeSWITCH 节点负责处理实际的通话，RTP 媒体流仍然在用户 UA 和 FreeSWITCH 之间直接传送[4]。

---

[1] LVS 的全称是 Linux Virtual Server，它是一个在 Linux 内核中实现的高度可伸缩、高可用的负载均衡服务器，详见 http://www.linuxvirtualserver.org/。

[2] 不过，这里一般不能像 HA 那样，如果一个计算节点失败，则将它正在计算的任务转给另外的服务器。对于 Web 服务来讲，如果客户在上浏览网页时不幸遇到这么一个坏的节点，则重新刷新一下页面就行了，对于 VoIP 应用来讲，则可以重新打一次电话。当然，对于 FreeSWITCH 来讲，比较终极的应用还可以是对于后端的每个节点都改成两台机器配对的 HA 方案，或者配置 N+M 方案。

[3] 典型的，在生产应用中，OpenSIPS 也应该用双机做 HA，以避免单点故障。同理，后面的数据库也应该有相应的 HA 机制保证可靠。关于 OpenSIPS 及数据库的 HA 超出了本书的范围，在此我们假设这些服务是可靠的。

[4] 当然 OpenSIPS 也有配合 RTPProxy 的 RTP 代理方案，在此我们就不讨论了。

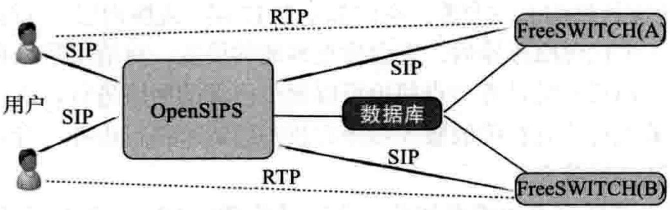


图 15-15 OpenSIPS 与 FreeSWITCH 负载均衡示意图

OpenSIPS 是著名的 SIP Proxy，它的前身是 OpenSER，而 OpenSER 的前身是 SER（SIP Express Router）。后来，由于版权的原因 OpenSER 更名为 OpenSIPS，而同时开发团队中的一部分人则开发出了另外一个分支，这一分支称为 Kamailio。我们无意讨论它们之间的差别与优劣，总之它们的根是一样的，配置使用起来也差不多。

1. 配置 FreeSWITCH

在配置 OpenSIPS 之前，我们应该配置两台一样的 FreeSWITCH（A 和 B），分别测试注册和打电话并保证都没问题。然后，两个 FreeSWITCH 要连接一个共享的数据库，数据库可以参照 12.3.2 节中的 ODBC 配置，也可以使用原生的 PostgreSQL 驱动。简单起见，我们把数据库放到跟 OpenSIPS 相同的机器上，当然数据库也可以放到独立的数据库服务器上。

重要的一点是，两台 FreeSWITCH 要连接同一个数据库，并且设置 vars.xml 中的 domain 参数为一个同一个名称，如 sip.example.com，将该 DNS 指向 OpenSIPS 服务器的 IP 地址。

修改 domain，将 vars.xml 中的

```
<X-PRE-PROCESS cmd="set" data="domain=$${local_ip_v4}"/>
```

改为：

```
<X-PRE-PROCESS cmd="set" data="domain=sip.example.com"/>
```

所以，这里需要用到 DNS 服务器支持。如果没有 DNS 服务器，也可以尝试将这里的 domain 设为 OpenSIPS 服务器的 IP 地址，或者使用 15.7.1 节介绍的 Domain 知识进行相关配置。

2. 安装配置 OpenSIPS

笔者在实验的时候用的是 OpenSIPS 1.8.2 版，如果版本号不同，配置文件也略有出入，其中“#”是注释起始符。

OpenSIPS 的安装应该很简单，笔者在这里使用的是 PostgreSQL 数据库，如果你使用 MySQL，也可以进行相应的替换。

使用如下命令进行 OpenSIPS 的安装：

```
cd opensips-1.8.2-tls
make all include_modules="db_postgres"
make include_modules="db_postgres" prefix="/usr/local" install
```

（1）创建数据库

可以使用 opensipsctl 命令创建数据库。在使用之前需要修改如下配置文件 /usr/local/etc/opensips/opensipsctlrc，以提供相应的参数。参数名称都很直观，在此就不多讲了，如：

```
# SIP_DOMAIN=opensips.org
DBENGINE=PGSQL
DBHOST=localhost
DBNAME=opensips
DBRWUSER=opensips
DBRWPW="opensips"
DBROOTUSER="opensips"
```

然后运行以下命令创建数据库：

```
# opensipsdbctl create
```

在配置好所有参数之前，我们先将 OpenSIPS 启动到前台，这样便于调试。配置文件及解释如下，首先是基本的参数：

```
debug=5                          # 输出详细的日志
memlog=1
fork=no                          # 启动到前台，fork=yes 则启动到后台
children=2                       # 在前台不起作用，如果 fork=yes，则控制启动多个进程
log_stderror=yes                 # 将 LOG 输出到标准错误（控制台）
log_facility=LOG_LOCAL0          # 日志输出，后台模式有用
disable_tcp=yes                  # 简单起见，我们只用 udp
disable_dns_blacklist = yes
auto_aliases=no
check_via=no
dns=off
rev_dns=off
listen=udp:192.168.1.118:7060    #监听的 IP 地址和端口，可以是多行，但前台模式只支持第一行
```

然后是加载模块配置，在这里，我们使用 OpenSIPS 中的两个模块来做负载均衡，一个是 dispatcher，它用于均衡注册消息；另一个是 load_balancer，它用于均衡呼叫相关的消息。其他模块的详细说明参见相关文档。

```
mpath="/usr/local/lib64/opensips/modules/" # 加载模块的路径

loadmodule "maxfwd.so"
loadmodule "sl.so"
loadmodule "db_postgres.so"
loadmodule "tm.so"
loadmodule "uri.so"
loadmodule "rr.so"
loadmodule "dialog.so"
loadmodule "mi_fifo.so"
loadmodule "signaling.so"
loadmodule "textops.so"
loadmodule "siptrace.so"
loadmodule "sipmsgops.so"                 # 1.8 版本以后才有
loadmodule "dispatcher.so"                # 注册消息均衡 (REGISTER)
loadmodule "load_balancer.so"             # 呼叫相关消息均衡 (INVITE)
```

下面的模块的配置参数。模块参数配置一般是用 modparam 来指定的，其中第一个参数

是模块名，第二个是参数名，不同的模块有不同的参数，具体如下：

```
# 默认的数据库的连接参数
db_default_url="postgres://opensips:opensips@localhost/opensips"
#fifo 模块的配置参数，其他模块类似
modparam("mi_fifo", "fifo_name", "/tmp/opensips_fifo")

modparam("dialog", "db_mode", 1)
<!-- modparam("dialog", "db_url", "postgres://opensips:opensips@localhost/opensips") -->

modparam("rr","enable_double_rr",1)
modparam("rr","append_fromtag",1)

modparam("siptrace", "trace_on", 1)

<!-- modparam("load_balancer", "db_url","postgres://opensips:opensips@localhost/opensips") -->

<!-- modparam("dispatcher", "db_url", "postgres://opensips:opensips@localhost/opensips") -->

# dispatcher 模块的配置参数
modparam("dispatcher", "ds_ping_method", "OPTIONS")
modparam("dispatcher", "ds_ping_interval", 5)
modparam("dispatcher", "ds_probing_threshhold", 2)
modparam("dispatcher", "ds_probing_mode", 1)
```

下面是路由的配置。路由的配置策略使用了类似C语言的脚本语言，看起来也比较直观，在此我们就不多解释了。路由配置内容如下：

```
route{
    if (!mf_process_maxfwd_header("10")) {
        sl_send_reply("483","Too Many Hops");
        exit;
    }

    if (!has_totag()) { # 初始请求
        record_route();
    } else { # 后续请求——遵循指定的路由
        loose_route();
        t_relay();
        exit;
    }

    if ( is_method("CANCEL" )) { # 处理 CANCEL 及重传
        if ( t_check_trans() ) t_relay();
        exit;
    }

    # 从现在开始，我们就只有初始请求了
    if (is_method("INVITE")) {
        # 如果是 INVITE 消息，则将调用 load_balance 函数选择一个可用的后端节点
        if (!load_balance("1","pstn","1")) {
            send_reply("503","Service Unavailable");
            exit;
        }
    } else if (is_method("REGISTER")) {
```

```
        # 如果是注册消息，则调用 dispatcher 模块的 ds_select_dst 函数分发请求
        # record_route();
        if (!ds_select_dst("1", "0")) {
            send_reply("503","Service Unavailable");
            exit;
        }
    } else { # 我们暂时不转发其他消息
        send_reply("405","Method Not Allowed");
        exit;
    }

    if (!t_relay()) { # 转发请求
        sl_reply_error();
    }
}
```

将上面的配置文件存为 lb.conf（名字和路径可以任意，反正我们是测试），然后运行如下命令启动 OpenSIPS：

```
# opensips -f lb.conf
```

接着就可以看到很多日志，启动另一个终端，在数据库中增加分发的目的服务器。opensipsctl 程序可以动态控制各模块的行为，如可以用如下命令在 dispatcher 中增加服务器：

```
# opensipsctl dispatcher addgw 1 sip:192.168.1.118 0 'FS1'
# opensipsctl dispatcher addgw 1 sip:192.168.1.119 0 'FS2'
```

然后启动一个 SIP 客户端注册到 sip.example.com 上，如果一切顺利则可以在 FreeSWITCH A 或 B 上收到注册消息。使用如下命令可以列出已注册的分机：

```
freeswitch> sofia status profile internal reg
```

由于两个 FreeSWITCH 都使用了同一个数据库，因此可以看到，在两台服务器上显示的信息都是一样的，就如同是在一台上一样[⊖]。

另外，上面我们说到，对于呼叫相关的消息，由 load_balancer 这个模块来处理，这里可以直接将负载均衡配置数据直接写入数据库：

```
psql> insert into load_balancer (group_id, dst_uri, resources, description)
values (1,'sip:192.168.1.118', 'pstn=32', 'FS1');

psql> insert into load_balancer (group_id, dst_uri, resources, description)
values (1,'sip:192.168.1.119', 'pstn=32', 'FS1');
```

其中，pstn=32 为服务器所能提供的资源，可以写成 'pstn=32;conf=20' 等，与 lb.conf 中 load_balance() 函数的第二个参数对应。

假设有 1000 和 1001 两个分机都已经成功注册。我们可以做一个测试，呼叫流程如下：

1）1000 发 INVITE 请求到 OpenSIPS，OpenSIPS 转发 INVITE 到 A，A 查询数据库，找到 1001 的 Contact 地址，由于 A 是一个 B2BUA，因此重新发起一个新的 INVITE 请求直

⊖ SIP 注册的流程是：SIP 客户端发 REGISTER 到 OpenSIPS，然后 OpenSIPS 发到 A 或 B，A 或 B 将 SIP 客户端的 Contact 地址写入数据库。由于数据库是共享的，因此在 A 和 B 上看起来都一样。

接到 1001 所在的地址，1001 接听后电话接通。

2）如果又有 1002 呼叫 1003，则 load_balance 模块会将 INVITE 路由到 B（因为 A 上已经有一路通话了），B 呼叫 1003，呼叫接通。

这样就达到了负载均衡的目的。

所有配置都测试无误后可以把 lb.conf 改到 /usr/local/etc/opensips.conf，并把其中的 fork 参数改为 yes 以使 OpenSIPS 启动到后台。

另外一个好处是，如果我们使用这种配置，可以随时添加或去掉一台 FreeSWITCH 服务器，而不影响使用。比方说在系统升级的时候，我们可以把所有的话务都定向到其中一台上，等待另一台负载为 0 后，进行升级。升级完成后，再将所有新的话务指到升级后的机器上，待旧机器话务降到 0 后即可升级旧的 FreeSWITCH。

在本例中，我们只是简单地介绍了 OpenSIPS 与 FreeSWITCH 配合使用的原理及基本配置，更多的示例可以参考 http://wiki.freeswitch.org/wiki/Enterprise_deployment_OpenSIPS，在此不再赘述。

15.12 压力测试

如果将 FreeSWITCH 应用于生产系统，一般需要经过一定的压力测试。下面我们讨论几种针对 FreeSWITCH 进行压力测试的方法以及相关参数和性能指标。

15.12.1 参数和指标

我们先来讨论一下我们关心的指标。当我们拿到一台 FreeSWITCH 服务器时，一般只关心两个问题——它最大支持多少用户？最大支持多少用户同时通话？

与传统的 TDM 交换机相比，FreeSWITCH 支持的最大用户数量几乎是无限的。因为，传统的 TDM 交换机中，每个用户（即每个电话）都会占用一个物理的硬件端口，这是一个绕不开的限制；而 FreeSWITCH 是 IP 的，所以基本上它仅受服务器内存及网络带宽的限制，可以很轻松地支持很大数量的用户。一般来说，FreeSWITCH 的本地用户即使不打电话也会周期性地发送 SIP 注册（REGISTER）请求，因而支持的最大用户的数量就可以认为是跟 FreeSWITCH 所能处理的注册消息正相关的。一般来说，SIP 客户端的注册失效（EXPIRE）时间是 1 小时，而根据 SIP 协议，客户端应该在半小时内重新发送注册消息，所以 FreeSWITCH 在 1 小时内仅需要处理一个用户的两次注册请求。假设 FreeSWITCH 在 1 秒内能处理 100 个注册请求的话，即它 1 秒就能服务 50 个用户，那么 1 小时就能服务 3600 × 50=18000 个用户。

当然，在 NAT 环境中，有的客户端为了保持 NAT 畅通，可能每十几秒就重发一次注册请求（实际上最好是重发 OPTION）。同样，FreeSWITCH 也可能 1 秒内能处理成千上万的用户注册。同时，具体的服务器以及网络环境也千差万别，因而，这些指标需要经过具体的测试才知道。

下一个指标就是最大并发数，也就是说一个系统最大可以支持多少人同时打电话。FreeSWITCH 是一个典型的 B2BUA，因而一般两个终端之间的通话就需要两个 Channel，而像会议或 IVR 之类的应用就是每个终端一个 Channel。简单起见，我们以系统并发的 Channel 的数量作为我们的并发指标。

与最大并发数相关的另一个指标是每秒能处理的呼叫数。一般来说，处理通话比较消耗资源的地方在于通话的建立和释放阶段——通话建立阶段需要处理 Session 的建立，媒体协商等；通话释放阶段需要处理 Session 的释放、写话单、清理现场等。而一旦通话建立后，双方互相收发媒体流，对系统资源的占用就可以认为是个常数了。

如果说系统每秒能处理 30 个呼叫（Channel）的建立与释放，每个呼叫持续 30 秒，那么系统的最大并发数就是 30 × 30=900，而系统每小时能处理的呼叫数就是 30 × 3600=108000。系统每小时所能处理的呼叫称为小时呼，又称 BHCA（Busy Hour Call Attempt，即最大忙时试呼次数），每秒能处理的呼叫称为 CPS（Call Per Second）。增加呼叫时长和增加每秒发起的呼叫数可以增加并发数。

在 FreeSWITCH 中，有两个参数可以限制系统能达到的最大呼叫量，以防止资源耗尽。其中，max_sessions 控制最大并发数，它的默认值是 1000；sps 控制最大的每秒呼叫量，默认值是 30。在系统中可以使用 status 命令查看这两个值（参见 4.4 节）。也可以使用如下命令修改这两个值：

```
freeswitch> fsctl max_sessions 5000
freeswitch> fsctl sps 100
```

上述命令仅在当前环境中生效，如果要使修改永远生效，则可以修改配置文件 conf/autoload_configs/switch.conf.xml 中如下的参数：

```
<param name="max-sessions" value="1000"/>
<param name="sessions-per-second" value="30"/>
```

15.12.2 呼叫测试

一般来说，大家比较关注呼叫测试。下面我们就来做两个实验。

1. FreeSWITCH 对呼测试

首先，最简单的测试方法就是用两台 FreeSWITCH 对着呼。设置两台 FreeSWITCH 服务器，如 A 和 B，它们的 IP 地址分别为 192.168.1.A 和 192.168.1.B，则可以在 B 上使用如下命令呼叫 A，并在本端执行 echo App：

```
freeswitch> originate sofia/external/service@192.168.1.A:5080 &echo
```

注意，我们这里直接呼叫 A 的 5080 端口，以避免认证。当然，在 A 上的 Public Dialplan 中我们需要设置一个路由播放音乐：

```
<extension name="Load Test">
<condition field="destination_number" expression="^service$">
  <action application="answer"/>
  <action application="playback" data="local_stream://moh"/>
```

```
</condition>
</extension>
```

如果对于两台 FreeSWITCH 对呼的部署方式不熟悉的读者可以回过头去再看一下第 13 章的内容。

为了能方便的发起大量呼叫，我们在 B 上简单地写一个 Shell 脚本，具体如下：

```
#!/bin/bash
IP=192.168.1.A
CMD="bgapi originate sofia/external/service@$IP:5080 &eccho"

for f in `seq 1 10`; do
    for f in `seq 1 30`; do
       fs_cli -x $CMD
    done
    sleep 1
done
```

上述脚本使用两个 for 循环，内循环执行 30 次发起 30 个呼叫，然后停顿 1 秒，再次发起 30 个呼叫；外循环则控制总的循环次数。

这仅仅是一个简单的脚本，通话一旦建立便会一直处理通话状态，如果要控制通话的时长，可以在 A 的 Dialplan 中播放一个特定长度的文件，也可以在 playback 之前使用 sched_hangup App 设置多长时间后挂机，如：

```
<action application="sched_hangup" data="+60"/>
```

上述配置表示从现在开始，60 秒后挂断通话。当然，如果仅仅测最大呼叫量的话，也可以不要让它自动挂机，而是在测试完成后手工使用 hupall 命令挂机。

2. 使用 SIPP 进行测试

SIPP㊀是一个很好的 SIP 测试工具，它可以对 SIP 协议及服务器性能进行比较全面测试。这里我们仅讨论最简单的情况。

首先，我们还是对上一节的 FreeSWITCH 服务器 A 进行测试，SIPP 可以运行在 A 上，也可以运行在 B 上。它的使用方法很简单，只需要运行以下命令就可以了：

```
$ sipp -sn uac -r 1 -d 10000 -rtp_echo 192.168.1.A:5080
```

上述命令中，“-sn uac”表示使用 SIPP 默认的场景文件（它将作为一个 UAC，即 SIP 客户端）；“-r 1”表示每秒发一个呼叫请求；“-d 10000”表示每个请求持续 10000 毫秒（即 10 秒）；“-rtp_echo”表示将收到的 RTP 流原样发回去（相当于 FreeSWITCH 中的 echo App）；最后的参数表示发到 FreeSWITCH 服务器 A 的 5080 端口㊁。

上述命令运行后，可以通过键盘上的 1、2、3、4 等按键控制界面上显示的统计信息，包含呼叫次数，成功、失败次数及各种时长信息等。图 15-16 是笔者在一次测试过程中的截

㊀ 参见 http://sipp.sourceforge.net/。

㊁ 因为 5080 端口无需认证，所以测试起来比较方便。对 5060 端口进行测试需要配置认证支持，这比较复杂，在此我们就不多介绍了。

图，图中显示了在 20cps 的情况下，当前有 2001 个并发呼叫，目前已成功进行了 2137176 次呼叫。

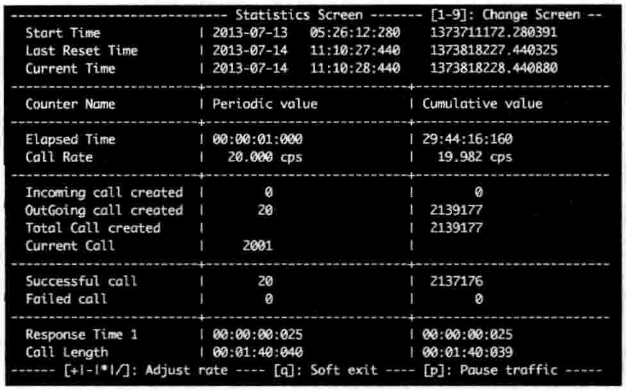


图 15-16　SIPP 测试截图

SIPP 使用 XML 描述的场景文件来描述何时该收、发什么样的 SIP 消息。不过其默认的配置文件少了点配置，因此 FreeSWITCH 社区推荐使用以下配置文件进行测试：http://www.freeswitch.org/eg/load_test/dft_cap.xml。下载完上述文件后，可以通过 -sf 参数指定使用该场景文件，如：

```
sipp -sf dft_cap.xml -r 1 -d 10000 -rtp_echo 192.168.1.A:5080
```

其中，“-r”表示每秒发一个请求，“-d 10000”表示每个呼叫持续 10000 毫秒（即 10 秒）；“192.168.1.21:5080”即 FreeSWITCH 的 IP 和端口；“-rtp_echo”表示我们把收到的 RTP 信息原样送回去，跟 FreeSWITCH 中的 echo App 类似。

SIPP 默认使用的被叫号码为 service，因此我们在上一节的 Dialplan 中就使用了 service，当然，也可以使用 -s 参数指定一个其他的被叫号码，如下列代码指定被叫号码为 load_test：

```
sipp -sf dft_cap.xml -s load_test -r 1 -d 10000 -rtp_echo 192.168.1.A:5080
```

当然，SIPP 还有好多选项，读者也可以研究一下，自己来写场景文件。更多的测试方法请参阅 SIPP 的有关资料，在此我们就不多讲了。

15.12.3　注册测试

SIPP 有一个场景文件可以支持注册测试，该文件可以在 http://sipp.sourceforge.net/doc/branchc.xml 处下载。有了该场景文件后，还需要配置一个数据文件，该文件提供注册用户的用户名密码等。如笔者使用的数据配置文件内容如下：

```
SEQUENTIAL
1001;[authentication username=1001 password=1234]
1002;[authentication username=1002 password=1234]
```

```
1003;[authentication username=1003 password=1234]
```

其中，“SEQUENTIAL”表示顺序执行。将上述文件存为 users.csv，然后就可以使用如下的 SIPP 进行测试了：

```
sipp -aa -sf branchc.xml -inf users.csv -r 10 -p 6060 192.168.7.102
```

其中，我们使用了“-inf users.csv”指定我们使用的数据文件；“-r 10”表示每秒注册 10 个；“-p 6060”指定 SIPP 监听的本地的端口号，它的默认值是 5060，明确指定端口号是为了防止当 SIPP 与 FreeSWITCH 在同一台主机上运行时产生冲突；最后的 IP 地址为 FreeSWITCH 的地址。

运行上述命令后，可以在 SIPP 中看到相关的统计信息，在 FreeSWITCH 中也可以使用“sofia status profile internal reg”命令查看客户端是否已正确注册。

15.12.4 编解码测试

FreeSWITCH 支持很多语音编码，不同编码的性能也不同。比如 G711（PCMU/PCMA）编解码基本上只要靠查表就行，所以是最高效的，而 G729 或 iLBC 之类的编码就需要比较高的 CPU。测试编解码的性能也很简单，如笔者在测试时在 FreeSWITCH 的 external Profile 中使用如下参数设置语音编码为 G729：

```
<param name="inbound-codec-prefs" value="G729"/>
```

然后在另一个（或几个）FreeSWITCH 服务器上向该被测试的 FreeSWITCH 上发起呼叫，或者使用 SIPP 对它进行呼叫。当有测试到来时，进入相关的 Dialplan 播放一个声音文件即可。

笔者曾在一台虚拟机上简单进行过测试，编解码的性能从高到低依次是：PCMU/A、G729（FreeSWITCH 官方商业版）、SILK、iLBC、iSAK，后面的几种编码需要比 PCMU 多使用几倍的 CPU。限于当时的环境没有测试其他编解码。当然，笔者的测试结果仅供参考，读者如果感兴趣可以使用这里的方法自己测试一下。

15.12.5 测试结果

在本例中，我们讨论了压力测试的一些相关指标并实验了几种测试方法。大多数的测试都没有标准答案，FreeSWITCH 官方也一直不正面回应压力测试的问题，理由是每个人的系统都是不一样的，业务场景也不同，操作系统以及软硬件环境也不一样，因此，测试的最终结果可能会差异很大。而且，压力测试能测试的呼叫流程和各种应用场景毕竟有限，所以最好也最准确的测试就是将 FreeSWITCH 应用到生产中去用实际的场景进行实地测试。

15.13 生产环境下的稳定性和安全性

如果将 FreeSWITCH 用于生产环境，或者用于公网上，那么稳定性和安全性是必须考

虑的。一般来说，FreeSWITCH 是比较稳定的，其默认的配置也是比较安全的。但真正的应用中肯定需要考虑稳定性的问题，而且仅使用其默认的配置也肯定不能满足应用要求。在此我们来简单介绍一下这两个问题。

15.13.1 稳定性

FreeSWITCH 本身是比较稳定的。它是很优秀的开源软件，全世界有很多人将它用于各种应用场景，因此，如果它有导致不稳定的地方，总会有人将问题反馈到社区，而且一般也会很快有人修复该问题。

另一方面来讲，FreeSWITCH 又是比较激进的，它的开发分支里很快会有大量的新特性加入，因此在测试不全面的情况下出现不稳定的情况也是难免的。

所以，如果用于生产环境，使用一个稳定的分支是有必要的。在本书截稿时，最新的稳定版是 1.2.22，最新的稳定分支是 1.2.stable。最新的 1.4 版目前还处于 Beta 阶段，相信很快就见到正式版了。

另外，即使稳定版也可能会崩溃。一旦出现这种情况，最好的解决方案就是立即恢复系统运行。在 UNIX 类系统上，可以使用 monit[㊀]等监控工具监控 FreeSWITCH 进程，并在崩溃后快速恢复 FreeSWITCH 运行。在对稳定性要求更高的场合，可以考虑使用我们前面讲过到双机 HA 方案或多机集群。

15.13.2 安全性

对于安全性而言，我们来考虑以下几个方面：

- FreeSWITCH 默认所有用户的密码都是 1234，这个一定是需要改的。可以修改 vars.xml 中的 default_password 全局变量，也可以针对每个用户在 conf/directory/default/*.xml 中修改。
- Event Socket（就是使用 fs_cli）默认监听在本地 Loopback IP 127.0.0.1 上，因此应该是安全的。但如果开了远程访问（如在其他主机上使用 Event Socket 与它连接）就记得修改一下这个密码。ESL 连接的密码传输都是明文的，因此不建议在公网上使用，除非在专用的 VPN 或加密隧道中。
- 默认情况下，mod_xml_rpc 是不加载的。如果你需要加载它，也应记住修改该模块的密码。它使用 HTTP Basic 来认证，因为密码都是明文传输的，不建议在公网上使用。如果需要使用的话建议在前面加 HTTPS 代理服务器。
- 默认的 5080 端口是允许任何人呼入的，因此需要确认它对应的 public Dialplan 里是否有最少的路由规则。
- 如果有黑客攻击，一般会发送大量的注册或呼叫请求暴力尝试所有可能的密码，可以考虑使用 Fail2ban[㊁]功能来抵御这种攻击。

㊀ 一个系系统监控工具，可以在检测到进程失败时迅速恢复运行，参见 http://mmonit.com/。

㊁ 失败即禁止，参见 http://wiki.freeswitch.org/wiki/Fail2ban。

- 在操作系统层，可以使用考虑使用 iptables 来设置合适的防火墙。另外，配合使用 Snort [1]或 AIDE [2]等防止入侵检测系统[3]也非常有用。
- 最后，最可怕的是 DDOS [4]攻击。这是一个世界性的难题，不过，如果你不幸遇到这种攻击了，那说明别人可能是找错了门，或者你的业务已经是世界级的大了。那时候你将可以有更多的资源来解决这一问题。不过为了防患于未然，一般来说选择一个好的数据中心能在一定程度上帮你解决这个问题。

总之，达到一定稳定性和安全性的同时肯定会带来一定的系统复杂性，另外也会增加一定的实施和维护成本，这是不可避免的。在本小节中，我们仅点出了一些应注意的问题，并提到了一些常用的软件和工具，希望对读者有所帮助。如果将 FreeSWITCH 真正应用于生产环境，肯定还有更多的问题需要考虑。更深入的讨论超出了本书的范围，在此我们就不再展开了。

15.14 小结

本章的内容比较广泛（主要是 FreeSWITCH 的功能确实比较多），从简单的企业通信应用，到传真、DTMF、电话连选功能、多租户配置等都有涉及。这些内容实现用起来比较有用，也通常是读者比较关心的问题。

通过使用组播方式进行通话的例子，让我们看到了通信功能的延伸，看到了 FreeSWITCH 功能的另一面，它不仅仅能打电话！

接下来，我们看到，通过与 Flash 及 WebRTC 技术的融合，FreeSWITCH 完全融入互联网了。

再后面，我也讨论了一些将 FreeSWITCH 应用于生产环境中需要考虑的 HA、大规模集群或分布式部署，以及相关的压力测试、稳定性及安全性问题。虽然我们没有将所有问题都给出答案，但对于想将 FreeSWITCH 应用于生产环境中的读者来说，这些内容肯定很有指导意义。

---

[1] 一个开源的网络入侵防御和检测系统，参见 http://www.snort.org/。

[2] Advanced Intrusion Detection Environment，即高级入侵检测环境，参见 http://en.wikipedia.org/wiki/Advanced_Intrusion_Detection_Environment。

[3] 简称 IDS，即 Intrusion Detection System，参见 http://zh.wikipedia.org/wiki/ 入侵检测系统。

[4] Distributed Denial of Service，分布式阻断服务攻击，又称洪水攻击，参见 http://zh.wikipedia.org/zh-cn/分布式阻断服务攻击。

第三部分

高级篇

本篇主要介绍与FreeSWITCH开发有关的内容。基于FreeSWITCH的开发一般有四种方式：

- 使用嵌入式脚本在FreeSWITCH内部灵活地控制呼叫流程，以及通过共享数据库或简单API等与现有业务系统集成；
- 使用Event Socket接口在外部程序中控制呼叫流程，控制更灵活，与其他系统也能更紧密地集成；
- 直接在FreeSWITCH中修改现有代码，或通过添加新模块以扩展FreeSWITCH的现有功能，并能结合前两种方式创建更强大的应用；
- 最后一种方式就是直接将FreeSWITCH的底层库libfreeswitch嵌入到其他系统中，这样被嵌入的用系统中就瞬间增加了FreeSWITCH的全部功能，这种方法用得比较少，但确实可用。

本篇我们会通过实际的例子，详细给大家讲解前三种开发方法，并对第四种方法做简单介绍。由于后两种方法需要熟悉FreeSWITCH的源代码，因此我们在介绍实际的例子之前也会教会大家怎么看源代码。总之，FreeSWITCH是开源的，如何更好地使用它仅取决于你的想象力。

读完本篇，你可以成为一名FreeSWITCH开发人员、软交换系统设计师、咨询师，以及无限可能师。

第 16 章
嵌入式脚本

FreeSWITCH 默认使用 XML Dialplan 配置呼叫流程。XML 文件描述性很强，因而也可以描述比较复杂的呼叫流程。但在一些比较高级的 IVR 应用和呼叫交互流程中，仅靠简单的 XML 的配置很难满足要求。因而我们还需要更灵活、更强大的解决方案。

除 XML Dialplan 外，FreeSWITCH 支持使用嵌入式的脚本语言控制呼叫流程。我们不仅可以用它们写出灵活多样的 IVR，给用户带来更好的体验，还可以通过它们很好地与你的业务进行无缝集成，以节省你的后台业务处理及管理成本。

在内部，FreeSWITCH 通过使用 swig [⊖] 工具来支持多种开发语言。简单来讲，swig 是一个包装工具（Wrapper），它可以将 FreeSWITCH 用 C 语言实现的一些功能包装成各种其他语言的接口、类或者方法，这样就可以在使用其他语言时以原生的方式调用。现在已知支持的语言有 C、Perl、PHP、Python、Ruby、Lua、Java、Tcl 以及由 Managed 支持的 .Net 平台语言，如 C#、VB.NET 等。FreeSWITCH 源代码中的 swig 脚本和程序已被转换成各种语言的接口了，因而开发者不需要安装 swig 工具就可以使用。这里有一点要专门指出，JavaScript 语言比较特殊一些，对它的支持是基于 Google 的 V8 库，在 mod_v8 模块中实现的。

使用程序语言来做前面提到的那些事情有两种方式：第一种是嵌入式脚本，第二种是独立的应用程序。如果使用后者，理论上讲，你可以使用任何你喜欢的语言，只要该语言支持 TCP Socket 就行。关于使用 Socket 方式来控制 FreeSWITCH 的编程方法我们将在 18 章讲解。本章主要关注嵌入式脚本。

16.1 FreeSWITCH 中的嵌入式脚本

其实前面我们学到的 XML Dialplan 已经体现了其非凡的配置能力，它配合 FreeSWITCH 提供的各种 App 使用时，也可以认为是一种脚本。当然，毕竟 XML 是一种描述语言，功能还比较有限，为了扩展其功能，FreeSWITCH 通过嵌入其他语言的解析器支持

⊖ 参见 http://www.swig.org 或 http://www.swig.org/translations/chinese/。

很多流行的编程语言。这些语言一般都能提供 if...else 判断及等循环跳转控制等，因而控制呼叫流程更加灵活。

一般来说，编程语言分为两种：编译型语言（如 C）和解释型语言（如 JavaScript、Perl 等）。使用解释型语言编写出来的脚本不需要编译，因而非常灵活方便。比较典型的、FreeSWITCH 支持的（可嵌入的）语言有[⊖]：

- Lua
- JavaScript
- Python
- Perl
- Java

一般来说，这些嵌入式脚本主要用于编写 IVR 类的应用，即主要用来控制一路通话的呼叫流程，虽然它们也可以控制多路通话（在后面我们也会讲到这样的例子，但这不是其擅长的）。

当然，这里说的一路通话不是说它们只能控制唯一一路通话。以 Lua 为例，你可以把呼叫路由到一个 Lua 脚本，当有电话进来时，FreeSWITCH 会为每一路通话启动一个线程，控制每一路通话的 Lua 脚本则在相应的线程内执行，互不干扰。Java 语言需要 Java 的虚拟机环境，比 Lua 要复杂些。

16.2 Lua

Lua 是一门小众语言，它可能不像其他语言（如 Java）那样“如雷贯耳”，但由于其优雅的语法及小巧的身段受到很多开发者的青睐，尤其是游戏开发人员[⊜]。

在 FreeSWITCH 中，Lua 模块是默认加载的。在所有嵌入式脚本语言中，它是最值得推荐的语言。首先它非常轻量级，mod_lua.so 经过减肥（Strip）后只有 272KB；另外，它的语法相对的简单。有人做过对比，在嵌入式的脚本语言里，如果 Python 得 2 分，Perl 得 4 分，JavaScript 得 5, 则 Lua 语言可得 10 分[⊜]，由此可见一斑。

另外，Lua 模块的文档也是最全的。笔者在使用其他语言（如 JavaScript）甚至写 Event Socket 程序时也经常参考 Lua 模块的文档。

注意

FreeSWITCH 完全内置了 Lua 解释器，因此与操作系统上的 Lua 版本以及各种第三方库

⊖ 其他脚本语言如 PHP、Ruby 等以前是支持的，由于它们有内存及性能问题，且没有志愿者维护，所以现在已经不支持了。

⊜ 我相信有很多人知道它是缘于 2010 年一则新闻，新闻中说：一个 14 岁的少年用 Lua 编出了 iPhone 上的名为 Bubble Ball 的游戏，该游戏下载量曾一度超过史上最流行的《愤怒的小鸟》。

⊜ Highly Embeddable: As far as embeddability goes - Python ranks a 2, Perl ranks a 4, JavaScript is a 5, and Lua is a 10! 参见：http://wiki.freeswitch.org/wiki/Mod_lua。

一点关系都没有，详细的配置信息可以参考 conf/autoload_configs/lua.conf.xml。

另外，自 2013 年 11 月中旬起，FreeSWITCH 中的 Lua 解释器升级到了 Lua 5.2 版本，因此，读者在写 Lua 脚本时要注意使用与 Lua 5.2 兼容的语法。而原来的 Lua 模块（基于 Lua 5.1）则被移动到了源代码目录下的 src/mod/legacy 目录中，如果需要 Lua 5.1 的用户可以自行编译该版本。

16.2.1 Lua 语法简介

Lua 的语法非常简洁易懂，有人说：如果你会其他编程语言，则在 30 分钟内就能学会它[⊖]。

大家都知道，如果有某种语言的编程经验，则学一种新的语言时可以对比着学。在这里，我们对比大家比较熟悉的 JavaScript 语言[⊜]来学习 Lua。

1. 相似性

Lua 与 JS（JavaScript 的缩写，下同）有很多相似的地方，简述如下。

- **变量无需声明**。Lua 与 JS 都是“弱”类型的语言（不像 C），所以它们不需要事先声明变量的类型。
- **区分大小写**。Lua 和 JS 都是区分大小写的。true 和 false 分别代表布尔类型的真和假，true 与 True 或 TRUE 是完全不同的。
- **函数可以接受个数不定的参数**。与 JS 类似，在 Lua 中，与已经声明的函数参数个数相比，实际传递的参数个数可多可少，如：

```
function showem( a, b, c )
print( a, b, c )
end

showem( 'first'  )                                --> first   nil     nil
showem( 'first', 'second' )                       --> first   second  nil
showem( 'first', 'second', 'third' )              --> first   second  third
showem( 'first', 'second', 'third', 'fourth' )  --> first   second  third
```

在 Lua 中，个数不定的参数列表使用“...”(称为 vararg expressions）来表示，如：

```
function showem(a, b, ...)
    local output = tostring(a) .. "\t" .. tostring(b)
    local theArgs = { ... }
    for i,v in ipairs(theArgs) do
        output = output .. "\t#" .. i .. ":" .. v
```

⊖ 找不到原始的出处了，好像是 FreeSWITCH IRC 上某个成员说的。不过，下面这篇博客倒是有点意思：http://ruoyusun.com/2013/03/23/pragmatic-lua-basics-in-30-mins.html。

⊜ JavaScript 是网页上浏览器端最流行的开发语言，而且随着 node.js 的流行，它在服务器端也迅速成长。当然，如果读者连 JavaScript 都不熟悉的话，也不要紧。JavaScript 的语法与 C 或 C++ 也比较类似。即使读者对所有这些语言都不熟悉，在这里两种语言对比着学习也没有什么坏处。本节大部分内容来自：http://phrogz.net/lua/LearningLua_FromJS.html 。

```
    end
    print(output)
end

showem('first' )                                   --> first  nil
showem('first', 'second')                          --> first  second
showem('first', 'second', 'third')                 --> first  second  #1:third
showem('first', 'second', 'third', 'fourth') --> first  second  #1:third  #2:fourth
```

- **哈希可以用方括号或点方式引用**。哈希表是编程语言中一种重要的数据结构。在 Lua 中，哈希表用 Table 来实现，在 JS 中，用稀疏在数组实现。无论如何，在两者中都可以使用如下的语法引用，如：

```
theage = gavin['age']
theage = gavin.age
```

- **数字区别不大**。在 JS 和 Lua 中，整数和浮点数是没有区别是的。它们在内部都是以浮点数表示。在 Lua 中，所有的数字类型都是 number 类型。
- **分号是可选的**。两者类似，在不产生歧义的情况下，行尾的分号可以有，也可以没有。不同的是对待分号的方式，在 JS 中，按惯例是包含分号的，而在 Lua 中，惯例是不包含分号的。
- **默认全局变量**。在 JS 中，如果用 var 声明一个变量并赋值，则它是本地变量；如果不用 var 声明，默认就是全局的，如：

```
function foo( )
{
    var jim = "This variable is local to the foo function";
    jam = "This variable is in global scope";
}
```

而在 Lua 中也类似，用 local 声明一个本地变量，省略 local 时则默认为全局变量：

```
function foo( )
    local jim = "This variable is local to the foo function";
    jam = "This variable is in global scope";
end
```

- **使用双引号和单引号表示字符串**。在 JS 和 Lua 中，字符串是用引号引起来的，并且单引号和双引号的作用没有任何不同[⊖]。引号要配对使用，但这两种引号可以混合以避免使用转义符，必要时可以使用 “\” 来转义，如（其中，箭头后面为实际变量的值）：

```
local book = "Seven's Book";                --> Seven's Book
local book = 'Seven\'s Book';               --> Seven's Book
local book = '"Awsome" book of Seven';      --> "Awsome" book of Seven
local book = "\"Awsome\" book of Seven"; --> "Awsome" book of Seven
```

- **函数是一等公民**。在 JS 和 Lua 中，函数是一等公民，这意味着，你可以将它赋值给

⊖ 这一点与 PHP、Ruby 以及 Bash 不同。在这三种语言中，双引号表示引用的内容中如果有变量的话，则要进行变量替换；而单引号表示引用的内容中的的字符串形式要保持原样，不管有没有变量。

一个变量，将它作为参数传递，或者直接加上括号进行调用，如在 Lua 中：

```
1    mytable = { }
2    mytable.squareit = function( x )
3        return x * x
4    end

5    thefunc = mytable.squareit
6    print( thefunc( 7 ) ) --> 49
```

其中，第 1 行声明一个 Table 类型的变量 mytable；第 2 ~ 4 行定义一个匿名函数，并将它赋值给 mytable 的 squareit 成员变量；第 5 行将上述成员变量的值又赋值给了一个变量 thefunc。至此，thefunc 实际上就代表第 2 行至第 4 行定义的匿名函数。最后，在第 6 行就可以通过 thefunc 引用该函数，该函数对输入的参数 7 进行平方计算（x * x），最后得到结果 49。

- **函数都是闭包**。在 JS 和 Lua 中，函数都是闭包。简单来说，这意味着函数可以随时访问该函数在定义时可以访问的本地变量，尽管在以后调用时这些本地变量逻辑上已经“失效”了。比如下面的 Lua 例子中，inBaseValue 中在定义时有效，却能在调用时照样使用它。

```
function MakeAdder( inBaseValue )
    return function( inValueToAdd )
        return inBaseValue + inValueToAdd
    end
end

add10 = MakeAdder( 10 )
add30 = MakeAdder( 30 )
print( add10( 1 ) )        --> 11
print( add10( 6 ) )        --> 16
print( add30( 3 ) )        --> 33
```

2. 区别

Lua 与 JS 又有很多区别，简述如下：

- **单行和多行注释不同**。JS 使用“//”做单行注释，而 Lua 使用“--”。JS 使用“/* ... */”来做多行注释，而 Lua 中使用“--[[...]]”（注意，这里的“...”表示实际被注释的内容）。

JS 中多行注释不能嵌套，解析器将在遇到第一个“*/”时终止该注释，而在 Lua 中，则可以使用类似“--[===[...]===]”的方式，通过在方括号中多加几个等号（前后等号的数量要匹配）来改变最外层的注释。参考以下 Lua 注释：

```
-- 本行是单行注释
local jim = "This is not commented"

--[[
local foo = "本行代码被注释掉了"
local bar = "本行代码也被注释掉了"
--]]

local jam = "本行是有效的"
```

```
---[[ 本行相关于一个单行注释
local foo = "本行也是有效的，为什么？"
local bar = "本行也是有效的，因为前面的注释中多了一个“-”
--]] 本行也相当于一个单行注释了

--[==[ 把下面这些全都注释掉
--[[
local foo = "foo"
local bar = "bar"
--]]
--]==]
```

- **用 end 终止程序块**。Lua 与 Ruby 类似，使用 end 来代替 JS 中的大括号来终止程序块，如下面是 Lua 中终止程序块的语法：

```
function foo( )
    --my code here
end

if foo( ) then
    --my code here
end

for i=0,1000 do
    --my code here
end
```

- **使用 nil 代表空值**。类似 Ruby，在 Lua 中，使用 nil 代表空值，在 JS 及 C 语言中则使用 null 或 NULL。

在 JS 中，空字符串（""）和 0 在条件测试中都为假（false）。而在 Lua 中，nil 和 false 是仅有的非“真”(True）值，其他所有的都表示测试结果为真。

- **Lua 中任何值都可以作为 Table 的键（Key）**。在 JS 中，对象（Object）的所有键都是字符串（如 myObj[11] 与 myObj["11"] 是相同的），而在 Lua 中，字符串、数字、甚至另一个 Table 都可以是键。参见如下 Lua 代码：

```
a = {}
b = {}
mytable = {}
mytable[1] = "The number one"
mytable["1"] = "The string one"
mytable[a] = "The empty table 'a'"
mytable[b] = "The empty table 'b'"

print( mytable["1"] ) --> The string one
print( mytable[1] )   --> The number one
print( mytable[b] )   --> The empty table 'b'
print( mytable[a] )   --> The empty table 'a'
```

- **Lua 中没有数组，任何复杂的数据类型都是 Table**。在 JS 里有明确的数组对象，并且有对应的操作数组的方法，如：

```
var myArray = new Array( 10 );  // 声明一个新数组，有 10 个空元素
```

```
var myArray1 = [ 1, 2, 3 ];       // 声明一个新数组，有3个元素
myArray1.pop();                   // 从数组中弹出最后一个元素
```

而在 Lua 中，对象是 Table、prototype 是 Table、哈希是 Table、数组是 Table、Table 是 Table、什么都是 Table。

Lua 中的所谓数组本身是一个 Table，它相当于 JS 里的稀疏数组，只是它的第一个元素的下标是从 1 开始的，而不是 0。可以使用 Table 的语法来创建数组。如下面 Lua 代码中的两种方法是等价的：

```
people = { "Alice", "Bob", "Carl" }
people = { [1]="Alice", [2]="Bob", [3]="Carl" }
```

如果拿 Table 当数组来用，可以使用两种方法来获取和设置数组的大小，并允许数组中有空值存在，如：

```
people = { "Alice", "Bob", "Carl" }
print( table.getn( people ) )                 --> 3

people[ 7 ] = "Someone Else "

print( table.getn( people ) )                 --> 3
print( people[ 10 ] )                         --> "Some Dude"

for i=1,table.getn( people ) do
    print( people[ i ] )
end
--> Alice
--> Bob
--> Carl

table.setn( people, 7 )
print( table.getn( people ) )                 --> 7

for i=1,table.getn( people ) do
    print( people[ i ] )
end
--> Gavin
--> Stephen
--> Harold
--> nil
--> nil
--> nil
--> Someone Else
```

- **数字、字符串以及 Table 都不是对象**。与面向对象概念里面的对象不同，数字、字符串以及 Table 本质上都不是对象。

Lua 仍然是面向过程的编程，大多数操作可以通过库函数实现，如：

```
print(string.len(mystring)  ) --> 11
print(string.lower(mystring)) --> hello world
```

其中，string.len 是一个函数，实际上 string 本身是一个 Table。从 Lua5.1 起，也支持一些类似面向对象的语法，如上面的例子等价于：

```
mystring = "Hello World"
print(mystring:len())  --> 11
print(mystring:lower())  --> hello world
```

在上面的例子中，通过使用“:”，把 mystring 看成一个对象，然后 len 和 lower 就类似于这个对象的方法。便实际上，Lua 是在内部把 mystring 这个假的“对象”特殊处理了，找到它所对应的数据类型（string）后，调用实际的 string.len()，并把 mystring 作为该函数的第一个参数，所以 mystring:len() 与 string.len(mystring) 是等价的。

- **没有 ++ 和 +=**。在 Lua 中没有 ++ 和 += 这样的缩写形式，所以变量自加必须用以下方式：

```
local i = 0;
i = i + 1;
```

字符串的拼接是使用“..”操作符实现的，如：

```
local themessage = "Hello"
themessage = themessage .. " World"
```

如果把一个字符串和一个数字相加，Lua 会试图将字符串转换成数字，如：

```
print(10 + "2")     --> 12
print(10 + "a")     --> 出错：attempt to perform arithmetic on a string value
```

- **没有三目运算符**。JS 或 C 中的“a ? b : c”语法是很贴心的，但在 Lua 中没有。不过，Lua 中有个“短路”语法，与该功能类似，如：

```
local foo = (math.random( ) > 0.5) and "It's big!" or "It's small!"

local numusers = 1
print( numusers .. " user" ..
(numusers == 1 and " is" or "s are") .. " online.")
--> 1 user is online.

numusers = 2
print( numusers .. " user" ..
(numusers == 1 and " is" or "s are") ..
" online.")
--> 2 users are online.
```

- **模式匹配**。正则表达式是很方便的字符串匹配工具。与 JS 以及其他语言不同，为了保持 Lua 的小巧且减少对其他库的依赖，Lua 没有使用常用的 POSIX(regexp)，也没有使用 PCRE（Perl 兼容的正则表达式），而是自己实现了模式匹配算法，其语法也有很大不同。

模式（Pattern）实际上就是 Lua 中的正则表达式。它与普通正则表达式最大的不同就是使用“%”而不是使用“\”来转义。Lua 模式中的字符分类如表 16-1 所示。

表 16-1 Lua 中的模式

模 式	说明
.	所有字符
%a	字母
%c	控制字符
%d	数字
%l	小写字母
%p	标点符号
%s	空白字符
%u	大写字母
%w	字母和数字连字符
%x	十六进制数字
%z	内部表示为 0 的字符

除此之外，它还有一些具有魔法含义的特殊字符，这些字符有：

```
( ) . % + - * ? [ ] ^ $
```

它们的含义都与 PCRE 里的差不多的，其中“-”的含义与“*”类似，都是重复前一个字符 0 次或多次，不同的是，“*”为最大匹配，它会尽量匹配更长的字符串，而“-”为最小匹配，它会尽量匹配更短的字符串。

3. 其他

Lua 的介绍已经占了本章很大的篇幅了。相信这些内容对于编辑简单的脚本也够用了。最后推荐一本学习 Lua 的书——《Lua 程序设计（第二版）》(原名 *Programming Lua*)。另外，英文好的读者可以参考另一本书 *Programming in Lua*，该书第一版是免费的，见 http://www.lua.org/pil/。

16.2.2 将电话路由到 Lua 脚本

在快速测试的情况下，可以直接使用如下的 originate 命令在 FreeSWITCH 中外呼一路通话，并在电话的本端执行 Lua 脚本：

```
originate user/1000 &lua(test.lua)
```

其中，lua 是一个 App，它的参数就是脚本的名字，脚本的默认路径在安装路径的 scripts 目录下，当然你也可以指定一个绝对路径，如 /tmp/test.lua。

在 Dialplan XML 中，使用下列配置便可将进入 Dialplan 的电话（Channel）交给 Lua 脚本接管。

```
<action application="lua" data="test.lua"/>
```

当然，除此之外，也可以直接使用 uuid_transfer 命令直接配合 inline Dialplan 将一个 Channel 路由到 Lua 脚本，如：

```
uuid_transfer <uuid> lua:/tmp/test.lua inline
```

总之，这里的 Lua 是一个标准的 App，在任何可以使用 App 的地方都可以使用它（如上面介绍的各种场景，以及后面要介绍的 Event Socket 等。甚至在 Lua 脚本中也可以再次使用 lua App 来调用下一个 Lua 脚本）。

16.2.3 Session 相关函数

在 Lua 环境中，FreeSWITCH 会自动生成一个 session 对象（实际上是一个 Table），因而可以在 Lua 脚本中使用 Lua 类似面向对象的语法特性编程，如以下脚本放播放欢迎声音[⊖]：

```
-- 应答
session:answer()
-- 睡一小会（毫秒）
```

⊖ 来自 Hello Lua: http://wiki.freeswitch.org/wiki/Mod_lua#Hello_Lua。

```
session:sleep(1000)

-- 播放声音文件
session:streamFile("/tmp/hello-lua.wav")

-- 挂机
session:hangup()
```

大部分与 Session 有关的函数都是跟 FreeSWITCH 中的 App 是一一对应的，如上面的 answer、hangup 等。有一点要特别说明：streamFile 对应 playback 这个 App。如果在 Lua 中没有对应的函数，也可以通过 session:execute() 函数来执行相关的 App，如 session:execute("playback", "/tmp/sound.wav") 与 session:streamFile("/tmp/sound.wav") 是等价的[⊖]。

需要注意，Lua 脚本执行完毕后默认会挂断电话，所以上面的 Hello Lua 例子中不需要明确的 session:hangup()。如果想在 Lua 脚本执行完毕后继续执行 Dialplan 中的后续流程，则需要在脚本开始处先设置不要自动挂机，语法如下：

```
session:setAutoHangup(false)
```

例如下列场景，test.lua 执行完毕后（假设没有 session:hangup()，主叫也没有挂机），如果没有 setAutoHangup(false)，则后续的 playback 动作得不到执行。

```
<extension name="test-lua">
    <condition field="destination_number" expression="^1234$">
        <action application="answer"/>
        <action application="lua" data="test.lua"/>
        <action application="playback" data="lua-script-complete.wav"/>
    </condition>
</extension>
```

下面，我们来讲与 Ssession 相关的几个常用的函数：

- getVariable。取得变量的值，如：

```
local moh = session:getVariable("hold_music")
local dest_number = session:getVariable("context")
local dest_number = session:getVariable("destination_number")
local cid_name = session:getVariable("caller_id_name")
local cid_number = session:getVariable("caller_id_number")
local ani = session:getVariable("ani")
local uuid = session:getVariable("uuid")
```

- getUUID。取得当前 Session 的 UUID，下列两行是等价的：

```
local uuid = session:get_uuid();
local uuid = session:getVariable("uuid")
```

- setVariable。设置通道变量，等价于 Dialplan App 里的 set：

```
session:setVariable("varname", "varval")
```

⊖ 实际上还是有一点区别：当使用 onInputCallback 回调时，session:execute 方式不会触发回调，而 session:streamFile 则可以。

❑ hangup。挂断当前通话：

```
session:hangup();
```

也可以同时指定挂断原因，如以下函数将在挂断时向对方返回用户忙：

```
session:hangup("USER_BUSY")
```

❑ ready。检查 Session 是否可正常使用，如果已经挂机就会返回 false。

在写脚本时，如果有循环，一定需要经常检测 session:ready() 是否为 true，否则 Session 挂机后 Lua 脚本可能仍然在死循环地运行。

```
while (session:ready() == true) do
    -- do something here
end
```

❑ streamFile：放音，相当于 Dialplan App 里的 playback。

```
session:streamFile("/tmp/test.wav")
```

❑ recordFile：录音，相当于 Dialplan App 里的 record，参数是：

```
file_name [,max_len_secs] [,silence_threshold] [,silence_secs]
```

其中，各参数含义如下：

❑ file_name：录音文件名。

❑ max_len_secs：录音最长的秒数。

❑ silence_threashold：一个声音阈值，如果声音小于该值，就认为是静音。

❑ silence_secs：如果静音时长大于一定秒数，则停止录音。

例如，以下函数将对当前的 Channel 录音，并存放到 /tmp/test_record.wav 中：

```
session:recordFile("/tmp/test_record.wav")
```

❑ read。类似于 Dialplan App 中的 read，用于播放一个声音并获取 DTMF。它的 5 个参数与 read 含义相同：

```
<min digits><max digits><file to play><inter-digit timeout><terminators>
```

6.4 节介绍的身份证号的例子用 Lua 改写如下：

```
digits = session:read(15, 18, "/tmp/input-id-card.wav", "5000", "#");
session:("log", "INFO ID Card Number: ".. digits .."\n");
```

读者可以发现 Lua 中的 read 比 Dialplan App 中的 read 少了一个参数。由于 session:read() 能返回值，因此那个参数就不需要了，实际收到的 DTMF 会返回到本例的 digits 变量中。

❑ playAndGetDigits。与 Dialplan App 中的 play_and_get_digits 类似，它的参数格式是：

```
<min_digits>, <max_digits>, <max_attempts>, <timeout>, <terminators>,<prompt_
audio_files>, <input_error_audio_files>,<digit_regex>, [variable_name], [digit_
timeout],[transfer_on_failure])
```

其中，大部分参数都很直观，也跟 play_and_get_digits 中类似。其中 timeout 是收齐所有号的超时值，而 digit_timeout 是允许的两次按键之音的时间间隔最大值，最后 transfer_

on_failure 指明如果失败后是否转到 Dialplan 中的一个 Extension 上去，它的格式应该是一个 Dialplan 三要素的格式串，如“failed XML dialplan”。

我们把 6.4 节中的身份证号的例子重写如下：

```
digits = session:playAndGetDigits(15, 18, 3, 10000, "#",
    "/tmp/input-id-card.wav", "/tmp/invalid_num.wav",
    "^\\d{15}|\\d{17}[0-9\\*]$")

session:execute("log", "INFO ID Card Number: ".. digits .."\n");
```

- **setInputCallback**。在放音或录音时，用户按下的 DTMF 可以用于触发一些功能。所以在这些状态下，Lua 支持如果收到 DTMF 等外部输入时，则调用相关的回调函数。setInputCallback 的作用就是设置（安装）一个回调函数。

```
01 function onInputCBF(s, type, obj, arg)
02     if (type == "dtmf") then
03         freeswitch.consoleLog("INFO",
04           "Got DTMF: " .. obj.digit .. " Duration: " .. obj.duration .. "\\n")
05           if (obj.digit == "3") then
06               return 'break'
07           end
08       end
09       return ''
10   end

11 session:setInputCallback('onInputCBF', '');
12 session:streamFile("local_stream://moh");
```

其中，第 1 行，我们定义了一个回调函数 onInputCBF，并在第 11 行使用 setInputCallback 将该函数绑定到该 Session 上。第 12 行执行 streamFile 实现一直不停地播放等待音乐。当用户按下话机上的键时，FreeSWITCH 就会回调上面指定的回调函数。下面我们来看一下该函数的内容。

回调函数一共有 4 个参数，在这里我们只关注 type 和 obj 两个参数。在此，由于我们的回调函数是由 DTMF 触发的，因此这里的 type 值为“dtmf”。obj 是一个 Table，它的两个 Key 分别是 digit 和 duration，分别表示 DTMF 的链值和时长。

上面的例子中第 2 行测试该回调是否是由 DTMF 引起，如果是，就打印相关 DTMF 信息。在第 5 行测试如果用户按了 3 的话，就返回 break。返回 break 会停止当前正在执行的 App（在这里我们正在播放保持音乐），如果后面没有其他脚本语句的话，Lua App 就会退出。如果返回空值（空字符串，如第 9 行），则它什么也不做。

如果我们顺序按 1、2、3，则上面例子在控制台上的输出结果如下：

```
[INFO] switch_cpp.cpp:1288 Got DTMF: 1
[INFO] switch_cpp.cpp:1288 Got DTMF: 2
[INFO] switch_cpp.cpp:1288 Got DTMF: 3
```

除 DTMF 外，回调还可能由其他的输入引起，如在进行语音识别的时候识别到相关关键词，这里我们就不多介绍了。

另外，使用 unsetInputCallback 函数也可以取消刚才设置的回调函数，在此我们就不多讲了。

更多与 Session 相关的函数可以参考相关的 wiki 文档：http://wiki.freeswitch.org/wiki/Mod_lua。

16.2.4 非 Session 函数和独立的 Lua 脚本

Lua 脚本中也可以使用跟 Session 不相关的函数，最典型的是 freeswitch.consoleLog()，其用于输出日志，如：

```
freeswitch.consoleLog("NOTICE", "Hello lua log!\n")
```

另外一个是 freeswitch.API()，允许你在 Lua 中执行任意 API，如：

```
api = freeswitch.API()
reply = api:execute("version", "")
freeswitch.consoleLog("INFO", "Got reply:\n\n" .. reply .. "\n")
```

上面 Lua 脚本可以直接在 FreeSWITCH 控制台上执行，如果将上述脚本保存到 /tmp/a.lua 中，则输出结果为：

```
freeswitch> lua /tmp/a.lua
2013-09-16 16:20:31.363305 [INFO] switch_cpp.cpp:1288 Got reply:

FreeSWITCH Version 1.5.6b+git~20130914T180606Z~60f5dec57e (git 60f5dec 2013-
09-14 18:06:06Z)
```

除此之外，其他的非 Session 函数还有 freeswitch.bridge()、freeswitch.email() 等，在此就不多介绍了。

非 Session 函数一般运行在独立的 Lua 脚本中。独立的 Lua 脚本可以直接在控制台终端上执行（使用 luarun），这种脚本大部分可用于执行一些非 Session 相关的功能（因为这里面没有 Session）。读到这里读者已经了解到了，Lua 是一个 App [⊖]，而 luarun 是一个 API。

我们刚刚讲到的 a.lua 就是一个典型的可独立运行的 Lua 脚本。独立运行的 Lua 脚本跟在 Dialplan 中用 Lua App 运行的不同，前者不会自动获得一个 session 对象（Table）。当然，独立运行的脚本也可以自行创建 session 对象。在 17.4 节可以看到相关的实例。

16.2.5 Event 相关函数

在第 5 章我们曾提到，FreeSWITCH 使用事件机制进行异步通信。在 Lua 脚本中，可以“生产”事件，也可以“消费”事件，下面我们来看一下与事件相关的函数。

FreeSWITCH 的事件也跟一个 SIP 消息类似，它包含一些事件头（Header）和可选的事件正文（Body）。在 FreeSWITCH 内部使用 C 语言结构体表示，可以序列化成类似 SIP 消息的简单文本格式（Plain）、JSON 或 XML。

- freeswitch.Event。初始化一个事件，该事件类型需要在 switch_event_types_t 枚举类型中有定义，它是在 switch_types.h 中定义的。如果使用了未定义过的名字，则统一为 MESSAGE。下面的例子初始化一个主事件：

⊖ 实际上，也有一个叫 Lua 的 API，它与 luarun 是等价的。不同的是，前者是在当前线程中执行的，而后者会新启动一个线程执行 Lua 脚本，并不阻塞当前线程的执行。

```
event = freeswitch.Event("MESSAGE_WAITING")
```

也可以初始化一个 CUSTOM 事件，其中第二个参数可以是任意字符串[⊖]，它将作为事件中的 Event-Subclass，如：

```
event = freeswitch.Event("CUSTOM", "freeswitch:book")
```

- event:addHeader。给事件增加一个事件头，如：

```
event:addHeader("MWI-Messages-Waiting", "no")
event:addHeader("MWI-Message-Account", "sip:1000@192.168.0.2")
event:addHeader("Sofia-Profile", "internal")
```

- event:fire。产生（生产）事件，如：

```
event:fire()
```

- event:addBody。给事件增加一个可选的正文，并使用 Content-Type 头标志正文的类型，如：

```
event:addHeader("Content-Type", "text/plain")
event:addBody("Hello FreeSWITCH")
```

- event:delHeader。从事件中删除一个头域，下面的例子可以替换 from 头：

```
event:delHeader("from")
event:addHeader("from", "1000@192.168.0.2")
```

- event:getHeader。在收到一个事件后，可以取得其头域的值，如：

```
event:getHeader("from")
event:getHeader("Caller-Caller-ID-Name")
```

- event:getBody。取得 Body 的值（如果有的话），如：

```
event:getBody()
```

- event:getType。取得事件的类型（名字），以下两种方法是等价的：

```
event:getType()
event:getHeader("Event-Name")
```

- event:serialize。将事件序列化成可读的形式（字符串），支持 plain text、JSON、XML 三种类型，如：

```
event:serialize()
event:serialize("json")
event:serialize("xml")
```

最后，我们来看一个完整的例子。该例子综合使用了上面介绍的各种函数，读者可以对照下面的代码和输出结果自行分析一下。代码如下：

```
function log(k, v)
    if not v then v = "[NIL]" end
    freeswitch.consoleLog("INFO", k .. ": " .. v .. "\n")
```

⊖ FreeSWITCH 内部也使用 CUSTOM 事件，如 sofia::register 等。使用时注意命名空间与内部不要发生冲突就行了，当然也可以选择跟内部一样的名字，用于伪造事件。

```
end

event = freeswitch.Event("CUSTOM", "freeswitch:book")
event:addHeader("Author", "Seven Du")
event:addHeader("Content-Type", "text/plain")
event:addBody("FreeSWITCH: The Definitive Guide")

type = event:getType()

author = event:getHeader("Author")

text=event:serialize()
json=event:serialize("json")
xml=event:serialize("XML")

log("type", type)
log("author", author)
log("text", text)
log("json", json)
log("xml", xml)

event:fire()
log("MSG", "Event Fired")
```

将上述内容保存到 /tmp/event.lua 中，执行结果如下（为了节省篇幅这里省略了某些事件头域及 JSON 和 XML 中的一些字段）：

```
freeswitch> lua /tmp/event.lua
[INFO] switch_cpp.cpp:1288 type: CUSTOM
[INFO] switch_cpp.cpp:1288 author: Seven Du
[INFO] switch_cpp.cpp:1288 text: 'Event-Name: CUSTOM
...
Event-Subclass: freeswitch%3Abook
Author: Seven%20Du
Content-Type: text/plain
Content-Length: 32

FreeSWITCH: The Definitive Guide'
[INFO] switch_cpp.cpp:1288 json: {
    "Event-Name":   "CUSTOM",
    "Core-UUID":    "bc647e68-47de-407f-b32a-d9bdf5c25786",
    "Event-Sequence":   "5000",
    "Event-Subclass":   "freeswitch:book",
    "Author":   "Seven Du",
    "Content-Type": "text/plain",
    "Content-Length":   "32",
    "_body":    "FreeSWITCH: The Definitive Guide"
}

[INFO] switch_cpp.cpp:1288 xml: <event>
<headers>
<Event-Name>CUSTOM</Event-Name>
<Core-UUID>bc647e68-47de-407f-b32a-d9bdf5c25786</Core-UUID>
<Event-Sequence>5000</Event-Sequence>
<Event-Subclass>freeswitch%3Abook</Event-Subclass>
<Author>Seven%20Du</Author>
```

```
<Content-Type>text/plain</Content-Type>
</headers>
<Content-Length>32</Content-Length>
<body>FreeSWITCH: The Definitive Guide</body>
</event>

[INFO] switch_cpp.cpp:1288 MSG: Event Fired
```

16.2.6 Chat 相关函数

FreeSWITCH 通过 mod_sms 支持文本消息。一个文本消息与一个 Session 类似，FreeSWITCH 收到文本消息后将执行 Chatplan，然后在 Chatplan 中可以执行 Lua 脚本。在 Chatplan 中的 Lua 脚本会自动获得一个 message 对象，该对象的内部表示跟 event 是一样的。因而与 Event 相关的函数，如 addHeader、delHeader、addBody、serialize 等，都是可以用的。除此之外，还有一个 chat_execute 函数，它可以执行 mod_sms 中支持的以下动作。

- fire：产生一个 MESSAGE 事件。
- send：发送消息。
- reply：回复消息。
- set：设置变量。
- info：显示信息。
- stop：停止消息路由。
- system：调用 system 函数执行系统调用。

下面的 Lua 脚本可以在 Chatplan 中执行，收到消息后先打印出来，然后修改目的号码和主机，并发送出去。

```
area_code = "010"
to_host = "192.168.0.2"

function log(k, v)
    if not v then v = "[NIL]" end
    freeswitch.consoleLog("INFO", k .. ": " .. v .. "\n")
end

log("Message", message:serialize())

to_user = message:getHeader("to")
message:delHeader("to")
message:addHeader("to", "internal/sip:" .. area_code .. to_user .. "@" .. to_host)

message:delHeader("to_host")
message:addHeader("to_host", to_host)

log("New Message", message:serialize())

message:chat_execute("send")
```

与在 Dialplan 中类似，在 Chatplan 中可以用以下方法调用 Lua 脚本，如：

```
<action application="lua" data="test.lua"/>
```

16.2.7 LUA 拨号计划

在第 6 章我们提到，拨号计划除 XML 外还有很多种，其中一种就是 LUA 拨号计划，即可以通过 Lua 脚本提供 Lua 风格的 Dialplan。

我们在这里讨论 LUA Dialplan，不仅可以增进对 Lua 脚本的理解，也可以增加对 Dialplan 的理解，即使读者永远不会用到 LUA 提供的 Dialplan，了解这些知识也是很有意义的。

下面的脚本入进路由阶段时将查询并生成一个 Dialplan，FreeSWITCH 接下来执行该 Dialplan，打印一些 Log，并执行 answer 和 playback。

```
function log(k, v)
    if not v then v = "[NIL]" end
    freeswitch.consoleLog("INFO", k .. ": " .. v .. "\n")
end

cid = session:getVariable("caller_id_number")
dest = session:getVariable("destination_number")

log("From Lua DP: cid:  ", cid)
log("From Lua DP: dest: ", dest)

-- Some Bussinuss logic here

ACTIONS =  {
    {"log", "INFO I'm From Lua Dialplan"},
    {"log", "INFO Hello FreeSWITCH, Playing MOH ..."},
    "answer",
    {"playback", "local_stream://moh"}
}
```

首先，跟在 Dialplan 中执行 Lua 脚本类似，这里也有一个 session 对象，可以执行所有与 Session 相关的函数，如获取主被叫号码（cid、dest）等。获取到相关信息后可以通过 Lua 相关的函数，如判断日期时间、连接数据库检查主被叫号码合法性及黑白名单等（这里我们省略了跟逻辑相关的操作）。最后，生成一个 Lua Table。该 Table 的名字必须是 ACTIONS。其成员可以是一个字符串或一个子 Table。FreeSWITCH 在 ROUTING 阶段获得该 Table 后，便可以进入 EXECUTING 阶段执行 ACTIONS 中定义的一系列动作（Action）。

Dialplan 有三个要素——Extension、Context 和 Dialplan 的名字，在 Lua Dialplan 中，Dialplan 的名字当然是 LUA，其 Context 就是 Lua 脚本的路径。把上面的脚本存为 /tmp/db.lua，我们可以立即使用 originate 测试。

```
    freeswitch> originate user/1000 test LUA /tmp/dp.lua
    [NOTICE] switch_channel.c:1030 New Channel sofia/internal/sip:1000@192.168.1.124:
34574 ...
    [NOTICE] sofia.c:5898 Ring-Ready sofia/internal/sip:1000@192.168.1.124:34574!
    [NOTICE] sofia.c:6561 Channel [sofia/internal/sip:1000@192.168.1.124:34574] has
been answered
    [NOTICE] switch_ivr.c:1831 Transfer sofia/internal/sip:1000@192.168.1.124:34574
to LUA[test@/tmp/dp.lua]
```

```
+OK bbe75fce-2e94-4730-94e0-060f1bc81026

[INFO] mod_lua.cpp:566 Processing ->test in context/script /tmp/dp.lua
[INFO] switch_cpp.cpp:1288 From Lua DP: cid:  : 0000000000
[INFO] switch_cpp.cpp:1288 From Lua DP: dest: : test
[INFO] mod_dptools.c:1588 I'm From Lua Dialplan
[INFO] mod_dptools.c:1588 Hello FreeSWITCH, Playing MOH ...
```

其中，originate 命令首先呼叫 user/1000，当它接听后，即转入 LUA Dialplan 中的 /tmp/dp.lua 这一 Context 进行路由，其对应的 extension 是 test。其执行结果也很直观。

用 originate 回呼是一种快速的测试方法，除此之外，读者也可以尝试在 XML Dialplan 中转入 LUA Dialplan，如：

```
<action application="transfer" data="$1 LUA /tmp/dp.lua"/>
```

或直接修改 Profile，让电话在呼入时直接进入 LUA Dialplan。如将 internal.xml 中的：

```
<param name="dialplan" value="XML"/>
<param name="context" value="public"/>
```

修改为：

```
<param name="dialplan" value="LUA"/>
<param name="context" value="/tmp/dp.lua"/>
```

执行 sofia profile internal rescan 可以使之生效。别忘了如果是注册用户拨打的话，还需要修改 User Directory 中的 user_context（因为它的优先级比 Profile 中的 context 要高），如在 1000.xml 中：

```
<variable name="user_context" value="/tmp/dp.lua"/>
```

总之，读者可以自己尝试一下，尽量完全掌握对这些不同的参数。如果遇到错误也不要紧，仔细看一下日志，应该很容易找到解决的办法。

16.2.8 连接数据库

在 Lua 中，可以使用 LuaSQL[⊖]连接各种关系型数据库，但 LuaSQL 与某些版本的数据库驱动结合有内存泄漏问题，配置起来也相对复杂一些。

另一种连接数据库的方式是直接使用 freeswitch.Dbh。它可以直接通过 FreeSWITCH 内部的数据库连接句柄来连接 SQLite 数据库或任何支持 ODBC 的数据库。下面是一个来自 FreeSWITCH Wiki 的例子，这里稍微做了一些改动：

```
-- 这里我们使用 SQLite 数据库，先初始化一个 DB Handle
local dbh = freeswitch.Dbh("/tmp/test.db")

-- 如果使用 ODBC 数据库，可以使用以下格式
-- local dbh = freeswitch.Dbh("odbc://dsn username password")

assert(dbh:connected()) -- 断言，测试数据库是否成功连接，如果连接不成功则退出
```

⊖ 参见：http://www.keplerproject.org/luasql/。

```
-- test_reactive 测试数据库表是否存在，它有三个参数：
-- 测试语句（查询，SELECT），删除语句（DROP）及创建语句（CREATE）
-- 意思是说，如果 SELECT 不成功，则先试图运行一个 DROP 操作，再用 CREATE 重新创建
dbh:test_reactive("SELECT * FROM test_table",
                  "DROP TABLE test_table",
                  "CREATE TABLE test_table (id INTEGER, name VARCHAR(255))")

-- 往我们新建的表中插入两条数据
dbh:query("INSERT INTO test_table VALUES(1, 'foo')")
dbh:query("INSERT INTO test_table VALUES(2, 'bar')")

-- 查询数据，并对查询结果集的每一行回调一个匿名函数（function），
-- 查询结果集将作为一个 Table 传入 row 参数，则实际列的值可以用 row.id、row.name 引用
dbh:query("SELECT id, name FROM test_table", function(row)
    freeswitch.consoleLog("INFO", string.format("Row #%s: %s\n", row.id, row.name))
end)

-- 更新表
dbh:query("UPDATE test_table SET name = 'changed'")

-- 打印更新行数（Affected Rows）
freeswitch.consoleLog("INFO", "Affected rows: " .. dbh:affected_rows() .. "\n")

-- 查看更新后的结果
dbh:query("SELECT id, name FROM test_table", function(row)
    freeswitch.consoleLog("INFO", string.format("Row #%s: %s\n", row.id, row.name))
end)

-- 释放 Handle，可选，默认用完后会自动释放
dbh:release()
```

执行结果如下：

```
[DEBUG] freeswitch_lua.cpp:352 DBH handle 0x7fa218a0fd28 Connected.
[DEBUG] switch_core_sqldb.c:1187 SQL ERR [no such table: test_table]
[SELECT * FROM test_table]
Auto Generating Table!
[DEBUG] switch_core_sqldb.c:1194 SQL ERR [no such table: test_table]
[DROP TABLE test_table]
[ERR] freeswitch_lua.cpp:391 DBH NOT Connected.
[INFO] switch_cpp.cpp:1287 Row #1: foo
[INFO] switch_cpp.cpp:1287 Row #2: bar
[INFO] switch_cpp.cpp:1287 Affected rows: 2
[INFO] switch_cpp.cpp:1287 Row #1: changed
[INFO] switch_cpp.cpp:1287 Row #2: changed
[DEBUG] freeswitch_lua.cpp:369 DBH handle 0x7fa218a0fd28 released.
```

我们也可以直接查看数据库（如果没有这个命令，需要安装 sqlite3 软件包，以“--”开头的注释为笔者所加）：

```
dujinfang@seven:/tmp$ sqlite3 test_db
SQLite version 3.7.12 2012-04-03 19:43:07
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> -- 显示所有表
sqlite> .table
```

```
test_table
sqlite> -- 显示表结构
sqlite> .schema test_table
CREATE TABLE test_table (id INTEGER, name VARCHAR(255));
sqlite> -- 查询
sqlite> select * from test_table;
1|changed
2|changed
```

16.3 其他脚本语言

虽然在嵌入式脚本中 Lua 是值得推荐的，但"萝卜白菜各有所爱"，并不是所有用户都选用 Lua 语言。而且，FreeSWITCH 作为一款优秀的开源软件也是尽量满足更多人的口味。因此，在这里作为对嵌入式语言的补充和比较，我们再来简单看一下另外几种语言。

16.3.1 JavaScript

相对于 Lua，大家可能对 JavaScript 更熟悉一些。JavaScript 是 Web 浏览器上最主流的编程语言，它最早用于配合 HTML 渲染页面，近几年由于 node.js [⊖]的发展使它在服务器端的应用也发扬光大。它遵循 EMCAScript 标准。

通过加载 mod_v8 [⊜]模块可以使用 JavaScript 解析器，该模块基于 Google 的 V8 [⊜] JavaScript 库。

我们在 16.2.1 节对 Lua 和 JavaScript 两种语言作了对比解析。在 FreeSWITCH 中，二者除了语法不同外，其用法类似。如使用 JavaScript（它是一个 App）执行一个 Session 相关的脚本，或 jsrun（它是一个 API）执行一个非 Session 相关的脚本。

例如，16.2.3 节的 Lua 脚本可以用 JavaScript 重写如下：

```
session.answer();
session.sleep(1000);
session.streamFile("/tmp/hello-js.wav");
session.hangup();
```

可以看出，除了与 Lua 在语法格式方面有点不同外，函数名都是相同的。但是，mod_spidermonkey 对应的 Wiki 页面上的函数参考说明也没有 Lua 详细。所以，如果使用 JavaScript 开发的话，读者可以使用 JavaScript 的语法，并参考 Lua 的 wiki 页面相关的函数说明。

可以在 XML Dialplan 中使用如下配置将来话交给上述脚本处理（假设文件名为 test.js）：

```
<action application="javascript" data="/tmp/test.js"/>
```

⊖ 参见：http://nodejs.org。

⊜ 参见 http://wiki.freeswitch.org/wiki/Mod_v8。注意，该模块仅是在最新的 1.4 版或 Master 分支上（2014 年 1 月 14 日以后）才有。对于以前的版本（如 1.2 版），请使用 mod_spidermonkey。mod_v8 由于使用了 V8 库，在性能上比以往有所提升，但两个模块提供的 API 是兼容的。

⊜ Google 的 V8 库用于在 Chrome 及 node.js 中以支持 JavaScript，参见 http://code.google.com/p/v8/。

在源代码目录中的 scripts/javascript 目录下，还有几个 JavaScript 应用的例子，读者可自行研究一下，在这里就不多介绍了。

16.3.2 Python

为了方便读者对各种语言做比较，我们这里再简单讲一下 Python 语言的例子。Python 是一门很流行的脚本语言，实际上，Linux 的安装程序的大部分脚本就是用 Python 写的。

Python 语言能很好地与其他系统集成，不过，它的可嵌入性比起 Lua 和 JS 来要差一些，因此，对编译 FreeSWITCH Python 模块的支持要稍微复杂一些。当然，这对于熟悉 Python 语言的读者来说也不算什么。因为，大部分在 FreeSWITCH 中选择使用 Python 的人都是因为对 Python 特别熟悉，否则的话，肯定就选择 Lua 了。

在编译 Python 模块前需要安装 Python 的开发包，笔者在 Ubuntu Linux 12.04 的安装过程如下：

```
# apt-get install python-dev
```

笔者使用的 Python 版本是 2.7，不过 Python 2.6 到 Python 3.x 的版本应该都可以支持。

安装了 Python 的开发包后，就可以直接安装 Python 模块了，在 FreeSWITCH 源代码目录中，执行以下命令就可以安装：

```
# make mod_python-install
```

如果安装过程中提示找不到 Python 之类的错误提示，则可以尝试重新配置并指定一个 Python 的路径，如：

```
# ./configure --with-python=/usr/bin/python     # 配置，指定 Python 的路径
# make && make install                         # 安装 FreeSWITCH 基本系统
# make mod_python-install                      # 安装 Python 模块
```

安装完成后会在 FreeSWITCH 的 mod 目录下生成 mod_python.so 模块，然后就可以在 FreeSWITCH 中加载该模块了，如：

```
freeswitch> load mod_python
```

与 Lua 和 JS 脚本不同。你可以将 Lua 或 JS 脚本放到任意路径中，但 Python 有自己的环境和 Package/Module 机制，因而需要配置特殊的路径。在笔者的 Ubuntu Linux 上，存放 Python 脚本的路径是 /usr/lib/python2.7/dist-packages/ ⊖。

FreeSWITCH 自带了一个 Python 的例子。我们可以将该例子复制到上述目录中。命令如下：

```
# cp src/mod/languages/mod_python/python_examply.py /usr/lib/python2.7/dist-packages/
```

然后，在 FreeSWITCH 控制台下就可以通过下列命令执行该脚本：

```
freeswitch> python python_example
```

执行命令后如果能看到如下输出就证明脚本执行成功了（脚本的实际内容读者可以自己

⊖ 不同的系统，Python 的路径可能不一样，如有的系统默认位置放在 /lib/python2.7/site-packages/ 目录中。

查看一下）。

```
freeswitch> python python_example
w00t!
'Event-Name: API
Core-UUID: 0611d6ee-1e7f-11e3-927e-bdbdad023004
FreeSWITCH-Hostname: localhost.localdomain
FreeSWITCH-Switchname: localhost.localdomain
FreeSWITCH-IPv4: 192.168.1.124
FreeSWITCH-IPv6: %3A%3A1
Event-Date-Local: 2013-09-16%2015%3A23%3A48
Event-Date-GMT: Mon,%2016%20Sep%202013%2007%3A23%3A48%20GMT
Event-Date-Timestamp: 1379316228983106
Event-Calling-File: switch_loadable_module.c
Event-Calling-Function: switch_api_execute
Event-Calling-Line-Number: 2333
Event-Sequence: 2523
API-Command: python
API-Command-Argument: python_example'
[NOTICE] mod_python.c:212 Invoking py module: python_example
[DEBUG] mod_python.c:281 Call python script
[DEBUG] mod_python.c:284 Finished calling python script
```

Python 模块加载成功后，我们就可以把前面的 Lua 和 JS 脚本重写为 Python 版的。内容如下：

```
import freeswitch

def handler(session, args):
    session.answer()
    session.sleep(1000);
    session.streamFile("/tmp/hello-python.wav");
    session.hangup();
```

其实各种语言都是相通的，函数的话用方法大同小异。

这里将上述脚本放到 /usr/lib/python2.7/dist-packages/book/test.py 中。另外，Python 环境还需要在当前的模块目录中有一个 __init__.py 文件，我们用以下命令生成一个空的文件即可：

```
# touch /usr/lib/python2.7/dist-packages/book/__init__.py
```

然后，就可以在 XML Dialplan 中通过如下的设置来调用上述脚本了：

```
<action application="python" data="book/test"/>
```

除此之外，其他脚本语言的使用也类似，读者可参照使用。值得一提的是，FreeSWITCH 有一个 mod_managed 模块支持 Windows .NET 架构下的语言（F#、VB.NET、C#、IronRuby、IronPython、JScript.NET），通过 mono 也可以支持其他平台（如 Linux）。

16.4 小结

本章讲了嵌入式语言并着重讲了 Lua 在 FreeSWITCH 中的用法，以及各种接口函数。

使用嵌入式语言可以非常方便地控制呼叫流程，创建丰富的 IVR 应用，通过 Lua 结合 Dbh 接口连接数据库还可以创建更强大的 IVR 流程。Lua 语言由于其天生支持嵌入式的天分，在这里展现了最好的表现。Java 需要 Java VM 的支持，Python 的嵌入能力不是非常好，Perl 比它们稍好一点但还是比 Lua 要多占一些资源。

Lua 模块功能实现得最完善。它不仅可以调用几乎所有的 API 和 App，还可以提供 Dialplan、XML 绑定等。

Lua 脚本运行于 FreeSWITCH 内部，它可以作为一个 App 来运行，相当于运行在一个 Channel 内部，这就使得它特别适合做 IVR 之类的应用，也就是说，它比较适合控制单腿的呼叫，一旦它执行 bridge 时 bridge 就阻塞了，因而不适合控制多条腿的情况。当然也不是说完全不能，Lua 也可以作为独立的与 Session 无关的脚本来运行，只是不是很常用。我们在第 17 章也会讲到 Lua 在各种使用场景下的实际例子。

此外，Lua 也适合用于配合“ execute_on_*”一族的通道变量使用，在 Channel 状态发生变化时执行相应的脚本。

当然，如果说嵌入式脚本还不能满足你的要求，那么就需要 Event Socket 粉墨登场了。我们将会在第 18 章讲到 Event Socket。

第 17 章
嵌入式及 HTTP 开发

上一章我们学习了嵌入式脚本的基本知识，这一章我们用它们来做些有用的事情。另外，除嵌入式脚本，我们还会在本章讲解一些配合 HTTP 服务器进行开发的例子，以便于读者了解更多的功能以及进行比较阅读。

17.1 用 Lua 脚本写个小游戏

工作再忙也总有时间轻松一下。好玩的游戏总会令人心情愉快。不过，在这里讲的这个小游戏不是我自己玩的，而是另有故事。在写作本书时，笔者的儿子刚刚 4 岁，他总会不时地过来敲打一下键盘，使得笔者总不能静下心来工作。当写到本章时，需要一个例子，笔者忽然来了灵感，就迅速给他写了个小游戏，这样他就可以自己玩一阵子。

由于是“迅速”写的，因此很简单。笔者给儿子在 iPad 上装了个软电话，他拨“1”就会进入 FreeSWITCH 上的一个 Lua 程序，该程序会提示他输入一个数字，并使用 TTS 读出这个数字。当然，如果只读“0123456789”肯定是太无聊了，所以笔者还加了一点点逻辑：如果输入的是“ * ”，就将数字减一，如果按的是“ # ”，就将数字加一。这样通过按“ # ”，数字就能一直累加上去。他能玩好一阵子，并乐此不疲。

后来，为了证明笔者确实收到了他的按键，笔者在程序中使用 system App 调用了系统的 banner [⊖]程序，把它输入的数字在控制台上打印出来。下面是数字“9”在笔者电脑上的输出（只是一直没意识到，在 Mac 平台上“banner”的输出是横着的）：

```
[INFO] switch_cpp.cpp:1293 DTMF: 9 Duration: 800
[NOTICE] mod_dptools.c:1973 Executing command: banner -w 40 9
```

⊖ UNIX 系统上自带的一个程序。它会使用“ASCII Art”形式输出字符。

```
                  #####
   #          ############
  ##         ##############
  #         ####        ###
  #         ##            ##
  #         #              #
  #         #              #
  ###        ##           ###
   #########################
     #####################
        ###############
```

当然，增加了 banner 以后又带来了一个副作用，那就是儿子隔一阵就跑过来看看笔者的 FreeSWITCH 控制台以验证他输入的是否正确，还是会影响工作。好了，闲话少叙，我们来看代码：

```
01 local x = 1

02 function onInput(s, type, obj, arg)
03   if (type == "dtmf") then
04       freeswitch.consoleLog("INFO",
05         "DTMF: " .. obj.digit .. " Duration: " .. obj.duration .. "\n")
06     if (obj.digit == "*") then
07       x = x - 1
08       if (x < 0) then x = 0 end
09       n = x
10     elseif (obj.digit == "#") then
11       x = x + 1
12       n = x
13     else
14       n = obj.digit
15     end

16     s:execute("system", "banner -w 40 " .. n)
17     s:speak(n)
18   end

19   return ''
20 end

21 session:set_tts_params("tts_commandline", "Ting-Ting")
22 session:answer()
23 session:speak("冰冰你好，请按一个数字")
24 session:setInputCallback('onInput', '')
25 session:streamFile("local_stream://moh")
```

程序的代码很简单。在第 2 行我们定义了一个 onInput 函数，当有按键输入时，系统会调用该回调函数，它用一个简单的算法计算一个变量值 n，然后在第 16 行调用 banner 在控制台上输出 n（其中的 s 变量就是传入的当前的“session”），并在第 17 行使用 TTS 技术“说”出 n 的值。

真正脚本的执行是从第 21 行开始的。该脚本在执行时会自动获得一个 session 变量，它唯一标志了当前的通话。在第 21 行，首先设置了我们将要使用的 TTS 的参数；然后在第 22 行进行应答；第 23 行播放一个提示音；第 24 行安装一个回调函数，当该 session 上有输入

时，它将回调该函数；第 24 行播放保持音乐。

为了能让他很简单地按“1”就呼叫到该脚本，笔者使用了下面的 Dialplan：

```
<extension name="Number Game">
  <condition field="destination_number" expression="^1$">
    <action application="lua" data="numbers_game.lua"/>
  </condition>
</extension>
```

当然，除此之外，笔者还实现了当他按“2”时可跟笔者来一阵视频通话：

```
<extension name="Video Me">
  <condition field="destination_number" expression="^2$">
    <action application="bridge" data="user/1007"/>
  </condition>
</extension>
```

等把书写完了，笔者一定好好想想，在 FreeSWITCH 中给他写更多更好玩的游戏。现在，就讲到这里吧。其实这个例子不仅仅是个游戏，它包含了在 Lua 脚本中接收 DTMF 输入、放音等基本操作，再多加上些 if...else 之类的语句，肯定能写出很强大的 IVR 应用了。

17.2 用 Lua 实现 IVR

用 Lua 可以创建很灵活的 IVR 应用，在此我们就来看一个电话充值的例子。

假设我们有一项电话自动充值服务，为会员提供自动服务，我们把这项服务称为空中充值服务。其中，每个会员都会有一个会员账号（以及密码），它可以根据账号查询余额，或者向该账号充值。如果需要充值，会员需要事先购买充值卡，在充值时需输入充值卡卡号来达到充值的目的。

为了提供比较好的用户体验，我们精心设计了一个充值流程，并画了一个简单的思维导图[⊖]，如图 17-1 所示。该图非常清晰地描述了用户查询及充值的流程。在此我们先不做太多解释，而是配合脚本代码讲解一下。

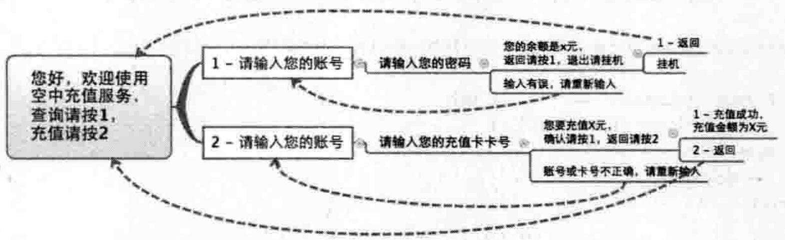


图 17-1　充值 IVR 流程思维导图

⊖ 思维导图即 Mind Map，是一种利用图像式思考辅助工具来表达思维的工具。在使用繁体中文的地区又称心智图，参见 http://zh.wikipedia.org/wiki/ 心智图。在解决此类问题时，大家也常用流程图。但流程图比较强调流程的正确性，而思维导图更强调思维（思路）的正确性。笔者一般倾向于使用后者，但不管使用前者还是后者，有一个图对于编程实现是非常有帮助的。

下面的代码首先是一行注释，接着设置了一些常量和变量：

```
01 -- Calling card charging demo. Author: Seven Du

02 error_prompt = "say: 输入错误，请重新输入"
03 account = ""
04 digits = ""
05 balance = 100   -- 余额
06 charge = 100    -- 充值卡上的金额
```

我们先定义一个 do_charge 函数，该函数有两个参数 account 和 charge，分别代表被充值的账号和充值的金额。在该函数中，我们没有做写数据库等操作，而是直接将欲充值的金额加到了全局变量中的余额（balance）上。代码如下：

```
07 function do_charge(account, charge)
08     balance = balance + charge
09     return balance
10 end
```

接下来定义一个主菜单函数。注意第 12 行，我们使用 session:ready() 函数判断该 session 是否还继续有效（即判断用户是否挂机），如果无效就立即返回。这是因为，我们的菜单可能会有多次循环，而在循环过程中如果用户挂机，并不会中止 Lua 脚本的执行，因此一定要保证有一定的退出机制。

第 13 行使用 playAndGetDigits 函数让用户选择是查询还是充值。第 15 行进行判断，如果用户输入有效，则将输入的数字（digits，值为 1 或 2）作为函数的参数传入 ask_account 函数中，并继续执行 ask_account 提示用户输入账号；当然，如果用户输入错误（最多 3 次，由 playAndGetDigits 函数保证），则直接跳到 goodbye 函数处理，我们下面会看到。main_menu 这一段代码如下：

```
11 function main_menu()
12     if not session:ready() then return end

13     digits = session:playAndGetDigits(1, 1, 3, 5000, "#",
           "say: 查询请按 1，充值请按 2", error_prompt, "^1|2$")

14     session:execute("log", "INFO main_menu:" .. digits)

15     if not (digits == "") then
16         ask_account(digits)
17     else
18         goodbye()
19     end
20 end
```

ask_account 函数用于询问用户的账号。注意类似第 22 行对于 session:ready() 的检测，基本上在每个函数中都有。另外，我们规定账号是 4 位的，因此在第 23 行，参数中允许用户输入的最小位长是 4，而这里我们使用最大位长是 5 的原因是，等待用户输入 4 位账号后，并不立即往下执行，而是等待用户按 # 号键。这里我们等待用户按 # 号键完全是一个用户习惯问题。实际上，如果有户不按 # 号键，我们这里的流程也是可以正常往

下执行的⊖。

当用户输入账号后，我们将其保存到 account 变量中。在第 29 行进行判断，如果在上一步用户选择了“1”，即查询，我们接着调用 ask_account_password（第 28 行）询问密码，否则（用户选择了“2”，充值）调用 ask_card（第 30 行）函数询问卡号进行充值流程。

```
21 function ask_account(service_type)
22     if not session:ready() then return end

23     digits = session:playAndGetDigits(4, 5, 3, 5000, "#",
           "say: 请输入您的账号，以井号结束", error_prompt, "^\\d{4}$")

24     session:execute("log", "INFO account:" .. digits)

25     if not (digits == "") then
26         account = digits
27         if (service_type == "1") then
28             ask_account_password()
29         else
30             ask_card()
31         end
32     else
33         goodbye()
34     end
35 end
```

ask_account_password 及 ask_card 函数的代码如下所示。它们的代码跟上面讲到的类似，都是收集了信息然后根据收到的信息调用相关函并进行处理。

```
36 function ask_account_password()
37     if not session:ready() then return end

48     digits = session:playAndGetDigits(4, 5, 3, 5000, "#",
         "say: 请输入您的密码，以井号结束", error_prompt, "^\\d{4}$")

39     session:execute("log", "INFO account p:" .. digits)

40     if not (digits == "") then
41         password = digits
42         check_account_password()
43     else
44         goodbye()
45     end
46 end

47 function ask_card()
48     if not session:ready() then return end

49     digits = session:playAndGetDigits(4, 5, 3, 5000, "#",
           "say: 请输入您的充值卡卡号，以井号结束", error_prompt, "^\\d{4}$")
```

⊖ 不过要等一会，读者可以自己试一下。也可以尝试把 5 改成 4 看看效果，那样的话就可以把语音提示里的“以井号结束去掉了”。

```
50     session:execute("log", "INFO card:" .. digits)

51     if not (digits == "") then
52         card = digits
53         check_account_card()
54     else
55         goodbye()
56     end
57 end
```

如果用户是在查询，则在用户输入密码后执行检查密码的函数（check_account_password）。在此我们直接使用硬编码（Hard coded）的“1111”（第 60 行）来验证账号和密码的合法性，在实际使用时可以配合查询数据库或第三方接口去检查密码的合法性。如果密码检查通过，则直接使用 TTS 技术播放余额（第 61 行），播放完毕后暂停 500 毫秒再返回主菜单（第 62 行）；当然，如果密码检查不通过，也进行相关提示并返回主菜单（第 65 ~ 66 行），以让用户有机会输入正确的值。

```
58 function check_account_password()
59     if not session:ready() then return end

60     if (account == "1111" and password == "1111") then
61         session:speak("您的余额是" .. balance .. "元")
62         session:sleep(500)
63         main_menu()
64     else
65         sesson:speak("输入有误，请重新输入")
66         main_menu()
67     end
68 end
```

至此，查询的代码分支就结束了，我们接着看充值的函数。第 71 行我们硬编码了卡号“2222”，如果用户输入正确，则在第 72 行语音提示用户将要充值的金额（充值卡上已有的金额，在第 6 行定义），然后让用户确定是否充值（第 73 行）。为了简单起见，我们没有再写新的函数，而是将逻辑判断都写到这一个函数里。代码如下：

```
69 function check_account_card()
70     if not session:ready() then return end

71     if (account == "1111" and card == "2222") then
72         session:speak("您要充值" .. charge .. "元")
73         digits = session:playAndGetDigits(1, 1, 3, 10000, "#",
               "say:确认请按1，返回请按2", error_prompt, "^[12]$")

74         if digits == "1" then
75             balance = do_charge(account, charge)
76             session:speak("充值成功，充值金额" .. charge ..
                "元，余额为" .. balance .. "元")
77             main_menu()
78         else
79             if digits == "2" then
80                 session:sleep(500)
81                 main_menu()
```

```
82              else
83                  goodbye()
84              end
85          end
96       else
87          session:speak("输入有误，请重新输入")
88          ask_account("2")
89      end
90 end
```

如果在上面的步骤中出错，我们会转到 goodbye 函数，该函数的定义如下（它只是简单地说声再见，然后挂机）：

```
91 function goodbye()
92     if not session:ready() then return end

93     session:speak("再见")
94     session:hangup()
95 end
```

上面是我们定义的函数，实际的脚本执行是从下面的代码才开始的。在本例中，为了使代码清晰，我们没有使用语音文件，而是使用了 TTS 技术。因此，第 96 行我们需要设置一下 TTS 的相关参数，该行可以保证 session:speak() 能正常放音，但经过测试，session:playAndGetDigits() 函数则不能正常放音，因此我们在第 97 和 98 行又分别设置了两个参数，保证其能正常放音。

```
96  session:set_tts_params("tts_commandline", "Ting-Ting");
97  session:setVariable("tts_engine", "tts_commandline")
98  session:setVariable("tts_voice", "Ting-Ting")
```

设置好这些参数后，我们就可以应答（第 99 行）、播放欢迎语（第 100 行），并开始执行主菜单函数（第 101 行）了。

```
99  session:answer()
100 session:speak("您好，欢迎使用空中充值服务")

101 main_menu()
```

在本例中，我们采用了一种类似状态机的设计，main_menu、ask_* 以及 check_* 等函数都可以看作是一个状态，根据输入的条件不同可以在这几个状态间转换，代码也比较清晰直观。而且，由于 Lua 的尾递归特性，在这些函数中可以循环地跳转也不会消耗栈空间。当然，我们在每个函数中都使用了 session:ready() 来判断当前的 session 是否有效避免了用户挂机后出现无限循环。

17.3　在会议中呼出

很多网友都曾问过，如何在会议中通过 DTMF 去呼叫其他人加入会议？实现的方法有很多种，这里我们来看一下如何通过配合 Lua 脚本来实现。

首先，我们来看下面的 Lua 脚本：

```
01 prompt="tone_stream://%(10000,0,350,440)"
02 error="error.wav"
03 result = ""

04 extn = session:playAndGetDigits(1, 4, 3, 5000,
    '#', prompt, error, "\\d+")

05 arg = "3000 dial user/" .. extn

06 session:execute("log", "INFO extn=" .. extn)
07 session:execute("log", "INFO arg=" .. arg)


08 if not (extn == "") then
09     api = freeswitch.API()
10     result = api:execute("conference", arg)
11     session:execute("log", "INFO result=" .. result)
12 else
13     session:execute("log", "ERR Cannot result=" .. result)
14 end
```

这个 Lua 脚本也很简单。首先，第 1 ~ 3 行初始化几个变量。然后在第 4 行执行 playAndGetDigits，它的用法我们在第 16 章已经讲过了，它首先播放 prompt 指定的声音（是一个拨号音），然后等待用户输入，最大收号长度是 4。如果用户输入错误，则播放 error.wav，并提示用户重新输入，最多可重试 3 次。最后收到的 DTMF 将存入变量 extn 中。

第 5 行，我们构造一个字符串，作为 conference 命令的参数，备用。

第 6 ~ 7 行，打印日志（**检查程序错误最好的方法就是多打印日志**）。

第 8 行，判断，如果输入的 extn 是一个有效的值，则在第 9 行初始化一个 api 变量，并在第 10 行执行一个 API 命令；如果用户输入的是 1006，则实际执行的命令就是 conference 3000 dial user/1006，它会主动外呼 1006 这个用户，被叫接听后便可进行会议。

不管呼叫成功还是失败，都会在第 13 行打印呼叫结果，如笔者在测试时，第一次拒绝了呼叫，日志如下：

```
[INFO] mod_dptools.c:1595 extn=1006
[INFO] mod_dptools.c:1595 arg=3000 dial user/1006
[INFO] mod_dptools.c:1595 result=Call Requested: result: [USER_BUSY]
```

第二次接听了呼叫，则日志结果如下：

```
[INFO] mod_dptools.c:1595 result=Call Requested: result: [SUCCESS]
```

当然，我们分析了上述脚本，还得研究一下怎么使用上述脚本。我们在 12.5.1 节曾讲过，在会议中，可以使用 DTMF 进行控制。我们先把会议控制中的 call-controller 部分的 * 和 # 号键对应的功能修改一下，让 * 号键对应执行我们刚写的 Lua 脚本（conference_dial.lua)，并把 # 号键对应的功能注释掉，以防止产生冲突。autolocal_configs/conference.conf.xml 中对应的配置如下：

```
<caller-controls>
<group name="default">
```

```
        <control action="execute_application" digits="*" data="lua conference_
dial.lua"/>
        <!-- <control action="hangup" digits="#"/> -->
    </group>
    </caller-controls>
```

然后，打个电话呼入名称为 3000 的会议，在会议中就可以通过按 * 号键在听到拨号音后输入一个号码进行外呼了。

如果只想会议管理员才能使用上述功能，也可以将上述功能键的映射关系放到单独的 group 中（如 group name="modrator"）并通过会议 Profile 中的 moderator-controls 指定该组以确保只有管理员才可以使用这些按键来进行控制。

17.4 一个在 FreeSWITCH 中外呼的脚本

曾有一个朋友这样问笔者：能否实现在 FreeSWITCH 中外呼，然后放一段录音？笔者的回答是：当然能！写个简单的脚本就行。但后来他要求还要知道呼叫是否成功，那实现就要复杂一点了。

笔者针对上边的问题很快写了一个例子。这个例子的实现思路是：将待呼号码放到一个文件中，每个号码一行，然后用 Lua 依次读取每一行，并进行呼叫。呼通后播放一个声音文件，并将呼叫（通话）结果写到一个日志文件中。

该脚本比较长，我们一段一段地来看。

首先设置一些常量和变量。第 1 行，设置呼叫字符串的前面的参数，其中，ignore_early_media 是为了保证对方应答后才开始播放（否则，在对方传回铃音或彩铃时播放就会开始，这不是我们想要的）；第 2 行指定我们呼叫的数据文件；第 3 行指定呼通后要播放的声音文件路径；第 4 行指定日志文件的路径。代码如下：

```
01 prefix = "{ignore_early_media=true}sofia/gateway/gw1/"
02 number_file_name = "/usr/local/freeswitch/scripts/number.txt"
03 file_to_play = "/usr/local/freeswitch/sounds/ad.wav"
04 log_file_name = "/usr/local/freeswitch/log/dialer_log.txt"
```

在脚本中，我们经常需要打印一些日志，简单起见，包装一个函数 debug，它实际上就是调用了 freeswitch.consoleLog 来打印日志的。代码如下：

```
05 function debug(s)
06     freeswitch.consoleLog("notice", s .. "\n")
07 end
```

接下来，我们定义一个呼叫函数 call_number，它接收一个号码（number）作为参数（第 8 行）。freeswitch.Session 会呼叫一个号码（第 11 行），并一直等待对方应答，对方应答后，将得到一个 session 变量；如果该 session 是正常的（第 12 行），则在第 13 行先等待 1 秒（1000 毫秒），以防止丢失最开始的声音。然后，在第 14 行使用 streamFile 播放一段声音，并在第 15 行挂机。第 18 行将进行忙等待；最后，第 22 行，函数返回挂机原因（hangup_cause）。

```
08 function call_number(number)
09     dial_string = prefix .. tostring(number);

10     debug("calling " .. dial_string);
11     session = freeswitch.Session(dial_string);

12     if session:ready() then
13         session:sleep(1000)
14         session:streamFile(file_to_play)
15         session:hangup()
16     end

17     -- waiting for hangup
18     while session:ready() do
19         debug("waiting for hangup " .. number)
20         session:sleep(1000)
21     end

22     return session:hangupCause()
23 end
```

实际的代码是从下面开始执行的。首先第 24 行打开存放电话号码的文件（准备读）和呼叫日志（第 25 行，准备写，追加）；然后进入一个死循环，第 27 行从数据文件中读出一个号码，放到 line 变量中；第 28 行测试，如果该行是空行或 nil 就说明文件读完了，跳出循环；第 29 行调用我们上面定义的 call_number 函数进行呼叫，并将结果存到 hangup_cause 变量中；最后，第 30 行将呼叫结果写到一个呼叫日志文件中。

然后是无限循环（while），每次读取一行，当读到空行或文件尾时，退出。

在 while 循环中，读到的每一行实际上是一个号码，调用上面定义的 call_number() 函数进行呼叫，并将呼叫结果写到 log_file 中。

```
24 number_file = io.open(number_file_name, "r")
25 log_file = io.open(log_file_name, "a+")

26 while true do

27     line = number_file:read("*line")
28     if line == "" or line == nil then break end

29     hangup_cause = call_number(line)
30     log_file:write(os.date("%H:%M:%S ") .. line .. " " .. hangup_cause .. "\n")
31 end
```

将上述脚本保存到 FreeSWITCH 的 scripts 目录中（通常是 /usr/local/freeswitch/scripts/），命名为 dialer.lua，然后在 FreeSWITCH 控制台上执行如下命令便可以开始呼叫了：

```
freeswitch> luarun dialer.lua
```

除此之外，还有一个 batch_dialer，用于批量外呼，感兴趣的读者可以自行研究一下：http://fisheye.freeswitch.org/browse/freeswitch-contrib.git/seven/lua/batch_dialer.lua?hb=true。

17.5 使用 Lua 脚本通过多个网关循环外呼

有时候，我们外呼需要通过多个网关。在 15.5.4 节我们曾讲到使用 mod_distributor 将呼叫分配到多个网关的方法。现在我们就来看一下如何用 Lua 脚本来实现上述功能。

首先看脚本的内容。到这里大家应该已经对 Lua 比较熟悉了，因此我们就不逐行讲了，仅将关键的地方讲一下。

第 3 行，我们定义了 4 个网关的名称，电话将通过这 4 个网关呼出。第 4 行我们定义了被叫号码取第一个从该脚本输入的参数，我们后面将会看到如何将这个值从 Dialplan 里传过来。

```
01 retries = 0
02 bridge_hangup_cause = ""
03 gateways = {"gw1", "gw2", "gw3", "gw4"}
04 dest = argv[1]
```

下面我们定义一个函数——call_retry，它将被递归调用。第 7 行中，每次递归我们都将 retries 参数加 1，这是为了在第 11 行选择一下个网关；第 8 行很关键，它判断电话是否挂断，如果电话已挂断就立即退出，防止过多递归。

```
05 function call_retry()

06     freeswitch.consoleLog("notice", "Calling [" .. dest .. "] From Lua\n");
07     retries = retries + 1
08     if not session.ready() then
09             return;
10     end

11     dial_string = "sofia/gateway/" .. gateways[retries] .. "/" .. dest;
12     freeswitch.consoleLog("notice", "Dialing [" .. dial_string .. "]\n");
```

通过上面的第 11 行中拼好了呼叫字符串后，下面第 13 行就调用 bridge 开始呼叫 b-leg 了，呼叫应该是阻塞的，直到呼叫失败或呼叫成功通话完毕后才会返回；然后在第 14 行我们能得到本次呼叫的结果，存放到变量 brige_hangup_cause 中；接下来判断呼叫结果，如果呼叫失败，而且失败原因是 16 ~ 19 行中的一种，并且重试次数小于 4（第 15 行），则暂停 1 秒（第 23 行）并重新调用 call_retry 函数进行呼叫。

```
13     session:execute("bridge", dial_string);
14     bridge_hangup_cause = session:getVariable("bridge_hangup_cause") or
session:getVariable("originate_disposition");
15     if (retries < 4 and
16         (bridge_hangup_cause == "NORMAL_TEMPORARY_FAILURE" or
17         bridge_hangup_cause == "NO_ROUTE_DESTINATION" or
18         bridge_hangup_cause == "CALL_REJECTED" or
19         bridge_hangup_cause == "INVALID_GATEWAY") ) then

20         freeswitch.consoleLog("notice",
21             "On calling [" .. dest .. "] hangup. Cause: [" ..
22             bridge_hangup_cause .. "]. Retry: " .. retries .. " \n");
23         session:sleep(1000);
```

```
24         call_retry();
25     else
26         freeswitch.consoleLog("notice", "Retry Exceed, Now hangup!\n");
27     end
28 end
```

值得注意的是，上面的 call_retry 函数是递归调用的。Lua 支持尾递归，所以递归时不会消耗栈空间，因此递归深度再大也没关系。不过，这里我们只有 4 个网关，故递归深度再大也不会超过 4。

上面只是定义了变量和函数，真正的函数是在下面开始执行的。第 30 行和第 31 行的两个变量设置也很重要，它保证在通过一个网关呼叫不通时电话不会挂断，以便继续尝试其他的网关；最后，第 32 行开始调用我们前面定义的 call_retry 函数进行外呼。

```
29 session:preAnswer();
30 session:setVariable("hangup_after_bridge", "true");
31 session:setVariable("continue_on_fail", "true");

32 call_retry();
```

在本例中，我们的 4 个网关是硬编码的（Hard Coded），在真正使用时也可以从 Dialplan 或其他地方传进来，在此我们尽量保持简单。

另外，从脚本中也可以看出，与 17.4 节讲的外呼脚本不同，该脚本是在 Dialplan 中使用的。我们可以用如下的 Dialplan 调用该脚本：

```
<extension name="Lua Multi-GW Example">
    <condition field="destination_number" expression="^0(.*)$">
        <action application="lua" data="call_gw.lua $1"/>
    </condition>
</extension>
```

其中，如果匹配到以 0 开头的被叫号码，我们“吃掉”0，把剩余的部分作为参数传给 Lua 脚本，然后在 Lua 脚本中就可以从 argv[1] 中获得这些被叫号码的值了（参见代码中第 4 行）。

17.6 在 FreeSWITCH 中执行长期运行的嵌入式脚本

前面我们讲过的 Lua 脚本都是“短暂”运行的——它们或者存在于一路通话的会话期内，或者是在命令行上执行一个短暂的命令。但在有些情况下，我们可能希望脚本能永远不停地运行，下面我们就来看一个例子。

几年前，笔者曾经写过一个 Lua 脚本，用于监控网关的状态。实现的思路是：如果接收到挂机事件，就判断该通话是否是经过一个网关出去的；如果是，就判断通话是否成功；然后记录统计结果，并将统计结果以几种方式呈现：

- 在 FreeSWITCH 中触发一个事件，由其他程序进行处理；
- 发送到一个远端的 HTTP 服务器上；
- 直接写入数据库；

❑ 其他方式，如写入一个文件等。

其他的程序在收到这些统计结果后再使用 Web 方式呈现，进而我们可以知道哪些网关（运营商提供的 SIP 中继）比较好，哪些网关总是出问题。

笔者是用 Lua 脚本实现的，我们需要让该脚本一直运行，因此首先想到的就是脚本中必须有一个无限循环。笔者尝试过这样做：

```
while true do
  -- Sleep for 500 milliseconds
  freeswitch.msleep(500);
  freeswitch.consoleLog("info", "blah...");
end
```

但这样的脚本是无法终止的⊖。由于 FreeSWITCH 使用 swig 支持这些嵌入式语言，而有些语言没有退出机制，所以所有语言的退出机制都没有在 FreeSWITCH 中实现，即使卸载（unload）相关的语言模块也不行。也是因为如此，为了避免产生问题，所有语言模块也都不能卸载⊜。

读者可能要问，既然是长期运行的脚本，那为什么要停止呢？是的，大部分时间是不需要停止的，但是我们都是开发人员，如果在开发过程中你需要调试和修改脚本，总不能每次都重启 FreeSWITCH 吧？

所以，后来笔者想了一个办法，通过使用事件机制构造另一个循环，然后就可以在检测到一个特殊事件后停止该循环。如下面的代码，其中，第 1 行，freeswitch.EventConsumer 定义一个事件消费者。第 2 行使用一个 for 循环不断从该事件消费者中取数据，pop(1) 是阻塞的，如果需要无阻塞的话，就可以使用 pop(0)。我们这里使用阻塞的方式，即直到有一个事件到来后它才执行循环体内的语句。当然，作为一个例子，我们在第 3 行仅打印收到的事件的内容。代码如下：

```
01 con = freeswitch.EventConsumer("all");
02 for e in (function() return con:pop(1) end) do
03     freeswitch.consoleLog("info", "event\n" .. e:serialize("xml"));
04 end
```

在 FreeSWITCH 中，事件总会有的。如每个电话初始化、挂机等都会有相应的事件。除此之外，FreeSWITCH 内部也会每 20 秒发出一个 HEARTBEAT 事件，这样你就可以根据该事件判断 FreeSWITCH 是否在正常运行，甚至还可以定时执行一些任务。

当然，到现在为止，还是不能运行上述脚本，因为它还是无法退出的。为了让它能退出，我们在上述代码的循环体中（第 4 行之前）再加入如下代码。这段代码的含义就是：当收到一个自定义的 CUSTOM 事件，并且事件的子类型是 lua::stopscript 时，便打印一个日志并退出（break）循环。代码如下：

⊖ 理论上，在循环体内通过检测一个 FreeSWITCH 全局变量的值也是可能终止的，不过笔者没有实验，而是采用了一种这里即将讨论的事件的方案。

⊜ 事实上，是在笔者发现该问题报告给 FreeSWITCH 社区后各模块才禁止卸载的。参见：http://jira.freeswitch.org/browse/FS-1365。

```
event_name = e:getHeader("Event_IVame")
event_subclass = e:getHeader("Event-Subclass")
if (event_name == "CUSTOM" and event_subclass == "lua::stopscript") then
    freeswitch.consoleLog("INFO", "--lua Script [" ..
        argv[0] .. "] got stop message, Exiting--\n")
    break
end
```

加入上述代码后，可以放心地去运行一下了。为了能让上面的脚本退出，我们可以再写一个脚本，来触发一个自定义的事件，如：

```
local event = freeswitch.Event("CUSTOM", "lua::stopscript");
event:addHeader("Exit-Reason", "Seven said you should stop");
event:fire();
```

解决了无限循环的退出问题后，你就可以在循环体内处理各种业务逻辑了。

脚本写好后，可以在控制台上使用 lua 和 luarun 命令执行该脚本。二者的不同就是 lua 是在当前线程中运行的，所以它会阻塞；而 luarun 会产生一个新的线程，不会阻塞当前的线程执行。

脚本后面可以加参数，如“ luarun test.lua arg1 arg2”，在脚本中，就可以通过 argv[1]、argv[2] 来获得参数的值。而 argv[0] 是脚本的名字。

另外，也可以将脚本配置到 conf/autoload_configs/lua.conf.xml 文件中，这样它就能随 FreeSWITCH 一起启动，如：

```
<param name="startup-script" value="test.lua"/>
```

最后，实际的例子可以参考笔者写的 gateway_report.lua 脚本，具体我们就不详细讲了，完整的代码在 http://fisheye.freeswitch.org/browse/freeswitch-contrib.git/seven/lua/gateway_report.lua?hb=true 处。

17.7 使用 Lua 提供 XML Binding

在 16.2.7 节我们介绍了用 Lua 提供动态拈号计划的例子，下面我们再看一下 Lua 能提供的另外一个功能——XML 绑定（Binding）。

前面我们学习过的 XML 配置文件都是静态的，在很多时候编辑静态的 XML 很不方便。FreeSWITCH 提供了一种机制可以在 XML 配置节点上绑定一些回调（函数或方法），然后当 FreeSWITCH 用到一段 XML 时，就可以调用该回调功能来取得 XML。我们可以使用 Lua 绑定一个回调，并通过 Lua 脚本来产动态的 XML 内容。

我们仍以 Dialplan 为例，注意这里我们使用的是 Lua 绑定 XML 提供的 Dialplan（提供的还是 XML Dialplan），而不是 LUA Dialplan。

首先，回想一下第 5 章，我们讲过 FreeSWITCH 配置文件 conf/freeswitch.xml 中有多个 Section，如 Configuraton、Directory、Dialplan、Chatplan、Languages 等，这里我们只绑定到 Dialplan。Lua 代码如下：

```
freeswitch.consoleLog("NOTICE", "SECTION " .. XML_REQUEST["section"] .. "\n")
xml = [[

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<document type="freeswitch/xml">
<section name="dialplan" description="RE Dial Plan For FreeSwitch">
  <context name="default">
    <extension name="9196">
      <condition field="destination_number" expression="^9196$">
        <action application="log" data="ERR I'm from Lua XML dialplan"/>
        <action application="answer"/>
        <action application="echo"/>
      </condition>
    </extension>
  </context>
</section>
</document>

]]

XML_STRING=xml
```

这段代码没有什么特别的，只是第 1 行打印了一行日志，该日志的内容包含从 XML_REQUEST 中取出来的参数。跟前面基于 Session 的脚本类似，所有这种绑定的请求，都会自动有一个 XML_REQUEST 变量，然后在 Lua 脚本中通过这个变量的内容来进行各种 if…else 判断，进而动态产生 XML。当然，本例子出于简单起见，在 Lua 脚本中还是写了一个静态的字符串。

脚本的最后一行，定义一个 XML_STRING 变量，当脚本执行完成后，FreeSWITCH 会在该变量中找到相应的 XML 文本并进行解析，以执行下一步的动作。

上述脚本编辑完成后，可以在 conf/lua.conf.xml 中进行如下设置并重启 FreeSWITCH 使之生效：

```
<param name="xml-handler-script" value="/tmp/dp_xml.lua"/>
<param name="xml-handler-bindings" value="dialplan"/>
```

该配置将 /tmp/dp_xml.lua 脚本绑定到 XML 配置文件中的 Dialplan 这个 Section，以后当 FreeSWITCH 需要查找 XML Dialplan 时，便会自动执行绑定的 Lua 脚本，生成动态的 XML。

17.8　语音识别

语音识别（Automatic Speech Recognition，ASR）算是一个高级话题。不过，之所以直到现在我们才讲到语音识别，并不仅是因为它高级，还因为使用它不像其他模块那样配置一下就能用了，而是需要配合一些脚本才能实现。

语音识别与 TTS 技术可以说是一对孪生兄弟，但“长相”却迥然不同。TTS 是把文字转成语音，而语音识别则是把声音转换成文字[⊖]。一般来说，TTS 技术是比较容易实现的，

⊖ 当然，它的定义本身不是这么狭隘，语音识别强调的是理解声音的内容，而不单是转换成文字。

最简单的实现仅需要用查表法将与文字对应的录音逐个查找到并读出，高级一些的借助一些语法和词法分析并借助语音合成技术能读出抑扬顿挫的声调；但语音识别就不同了，它不仅需要语法和词法分析，还需要“理解”声音的内容，以转换成合适的文字。

语音识别分为基于关键词的识别和自然语音识别。基于关键词的识别比较成熟，因为词汇数量有限，比较容易做到精确。这类应用一般用于声控场合，如发出打开、关闭（设备或程序）、呼叫（某人）等命令。基于自然语言的识别则比较难，多年来人们进行了大量的研究，但鲜有较好的应用。后来，随着 iPhone 上 Siri 的出现，语音识别好像一下子迎来了曙光，众多的语音识别产品也很快推出来了。

当然，在此我们无意关注这些产品，也不想深入探究这些底层的技术，而是想了解 FreeSWITCH 如何与语音识别搭上边——由于 FreeSWITCH 内部处理的媒体类型主要是声音，自然一切与声音有关的应用都能搭上边。

17.8.1 使用 Pocket Sphinx 进行中文语音识别

Sphinx[㊀]是卡内基梅隆大学研究的开源语音识别软件，Pocket Sphinx[㊁]则是一个轻量版，主要用于嵌入式及手持移动设备。FreeSWITCH 中有一个 mod_pocketsphinx[㊂]模块可以通过 PocketSphinx 进行语音识别。该模块不是默认安装的，但可以参照前面其他模块的安装方法，在源代码目录下使用“make mod_pocketshpinx-install”并在 FreeSWITCH 控制台上使用“load mod_pocketsphinx”命令加载。

1. 英文语音识别快速体验

FreeSWITCH 自己带了一个 Pizza 的例子，大意是打电话到一个比萨店，店里的 IVR 会自动询问你欲订何种比萨，你的回答对方可以识别出来。读者可以参阅该模块的 Wiki 页面看一下如何测试该例子。不过，该例子讲起来比较复杂，下面我们看一个简单点的例子。

首先，语音识别需要一个语法文件。在 FreeSWITCH 源代码目录下的 script 目录下有一个 yes_no.gram 的语法文件的例子，我们先把它复制到 FreeSWITCH 安装目录的 grammar 目录下。该文件不长，全部内容如下：

```
#JSGF V1.0;
/** JSGF Grammar for example */

grammar example;

<yes> = [ yes ];
<no> = [ no ];

public <results> = [ <yes> | <no> ];
```

该语法文件是根据 JSGF[㊃]语法写的，它的作用是对照字典组成相关的词语或句子。在

㊀ 参见 http://www.speech.cs.cmu.edu/sphinx/doc/Sphinx.html。

㊁ 参见 http://www.speech.cs.cmu.edu/pocketsphinx/。

㊂ http://wiki.freeswitch.org/wiki/Mod_pocketsphinx。

㊃ JSpeech Grammar Format，即 JSpeech 语法格式。参见 <http://www.w3.org/TR/jsgf/>。

此，该语法仅定义了“yes”和“no”，因此，它仅能识别英文单词“yes”或“no”。不过，对于我们演示语言识别的例子来说足够了。

接下来设置如下 Dialplan。这一次我们使用了 flite 作为 TTS，并使用 kal 嗓音。play_and_detect_speech 是一个 App，它的作用是播放一段声音（声音文件或 TTS 文本）并进行语音识别。它的第 1 个参数是 say:please say yes or no，用于指定播放的 TTS 文本；第 2 个参数是 detect:pocketsphinx，表示使用 pocketsphinx 进行语音识别；第 3 个参数 yes_no，指定一个语法文件（就是我们刚刚准备的那个）。当语音识别完成后，识别的结果将存入 detect_speech_result 通道变量中，同时使用 log App 将它的内容打印到日志中以便观察识别结果。完整的 Dialplan 设置如下：

```
<extension name="ASR">
    <condition field="destination_number" expression="^1234$">
        <action application="answer"/>
        <action application="set" data="tts_engine=flite"/>
        <action application="set" data="tts_voice=kal"/>
        <action application="play_and_detect_speech" data="say:please say yes
or no detect:pocketsphinx yes_no"/>
        <action application="log" data="INFO ${detect_speech_result}"/>
    </condition>
</extension>
```

设置好上述 Dialplan 后，拨打 1234 便进入了语音识别程序。它将首先播放“please say yes or no”，当说了“yes”后，控制台上打印出如下的日志：

```
[INFO] switch_ivr_async.c:3718 (sofia/internal/1012@192.168.7.6) START OF SPEECH
[INFO] switch_ivr_async.c:3708 (sofia/internal/1012@192.168.7.6) DETECTED SPEECH
[INFO] mod_dptools.c:1602 <?xml version="1.0"?>
<result grammar="pizza_yesno">
  <interpretation grammar="pizza_yesno" confidence="100">
    <input mode="speech">yes</input>
  </interpretation>
</result>
```

其中，第 1 行 START OF SPEECH 表示 PocketSphinx“听”到了有人说话；第 2 行 DETECTED SPEECH 表示识别出了说话的内容；从第 3 行可以看出，识别的结果是使用 XML 表示的，里面的 yes（在第 6 行）即表示识别到的文本。

2. 中文语音识别

在中国，语音识别软件必须能识别出中文才有意义。接下来我们再看一看中文语音识别的例子。

（1）准备工作

我们需要先选择一个中文的声音模型，笔者选择了 tdt_sc_8k，放到 grammer/model 目录下面；另外，还得有个字典文件，笔者选择了 zh_broadcastnews_utf8.dic，放到 grammar 目录下面。在 conf/autoload_configs/pocketsphinx.conf.xml 中进行如下配置：

```
<param name="narrowband-model" value="tdt_sc_8k"/>
<param name="dictionary" value="zh_broadcastnews_utf8.dic"/>
```

字典文件本身有点大，笔者精简了一下，内容大致如下：

```
杜 d u
金 j in
房 f ang
令 l ing
狐 h u
冲 ch ong
乔 q iao
布 b u
斯 s if
孙 s un
悟 w u
空 k ong
中 zh ong
山 sh an
杜金房 d u j in f ang
令狐冲 l ing h u ch ong
乔布斯 q iao b u s if
孙悟空 s un w u k ong
孙中山 s un zh ong sh an
```

在此我们就不深入研究字典文件的意义了，读者大致也能明白上面首先是中文的字词，后面是相关的语素就可以了。除了字典文件外，我们还需要有一个语法文件，存放在 grammar 目录下，不妨取名为 zh.gram。内容如下：

```
#JSGF V1.0;

/**
  * JSGF Grammar for pizza_size names
  */

grammar names;


<du> = 杜 ;
<jin> = 金 ;
<fang> = 房 ;
< 孙 > = 孙 ;
< 中 > = 中 ;
< 山 > = 山 ;

public <jf> = <jin><fang>;
public <djf> = <du><jin><fang>;
public <szs> = < 孙 >< 中 >< 山 >;
```

该语法文件也是根据 JSGF 语法写的。准备好这些内容后，我们就可以写我们的语音识别脚本了。

注意

需要指出，本例中用到的部分语言模型及数据字典不是默认安装的，语音识别是一个相对专业的技术，而使用 Pocketsphinx 进行中文语音识别也没有完整的文档，感兴趣的读者如果需要按这里的内容进行实践的话，可以根据这里提供的关键字自行到互联网上搜索相关资源。

（2）语音识别 Lua 脚本

我们还是以 Lua 为例来介绍语音识别的脚本。首先，我们当然要先写一个 log 函数，以便知道脚本在运行过程中都发生了什么。内容如下：

```
function log(text)
    freeswitch.consoleLog("INFO", text);
end
```

然后，我们写一个 onInput 函数，实现在有输入的时候可以得到回调。在前面的例子中我们学过，当有用户 DTMF 输入时便可以调用回调函数。在语音识别中，当识别到用户的语音时，也可以回调相应的函数。如下面函数的前几行，当回调是由 DTMF 引起的时（第 3 行），便打印一条日志（第 4 行）并退出（第 5 行）：

```
01 function onInput(s, type, obj)

02     log("Callback type:" .. type)
03     if (type == "dtmf") then
04         log("DTMF: " .. obj.digit .. "\n")
05         return "break"
06     end
```

当检测到语音识别的结果时，回调的类型（type）将是 event（第 7 行），obj 即是实际的事件，我们可以在事件中取出 Speech-Type 的值（第 8 行）。

当 Speech-Type 值为 begin-speaking 时，说明系统检测到你说话了，我们便打印出事件的内容（第 10 行）。

```
07     if (type == "event") then
08         local event = obj:getHeader("Speech-Type")
09         if (event == "begin-speaking") then
10             log("\n" .. obj:serialize())
11             return ""
12         end
```

在实际的运行中，输出结果如下，可以从中看出 Speech-Type：

```
Event-Name: DETECTED_SPEECH
Core-UUID: bf7f346e-25c8-498b-9a2c-8d0e5f70a266
FreeSWITCH-Hostname: seven.local
FreeSWITCH-Switchname: seven.local
FreeSWITCH-IPv4: 192.168.7.6
FreeSWITCH-IPv6: %3A%3A1
Event-Date-Local: 2013-11-21%2014%3A48%3A55
Event-Date-GMT: Thu,%2021%20Nov%202013%2006%3A48%3A55%20GMT
Event-Date-Timestamp: 1385016535584614
Event-Calling-File: switch_ivr_async.c
Event-Calling-Function: speech_thread
Event-Calling-Line-Number: 3860
Event-Sequence: 7705
Speech-Type: begin-speaking
```

当识别出语音时，Speech-Type 的值就是 detected-speech（第 13 行），然后我们在第 14 行将语音识别暂停（pause），代码如下：

```
13         if (event == "detected-speech") then
14             session:execute("detect_speech", "pause")
15             log("\n" .. obj:serialize())
```

同时我们看到，第 15 行打印了收到的事件的内容，实际运行效果如下（为节省篇幅，我们省略了一部分事件头）：

```
Event-Name: DETECTED_SPEECH
Speech-Type: detected-speech
Content-Length: 168

<?xml version="1.0"?>
<result grammar="zh">
  <interpretation grammar="zh" confidence="100">
    <input mode="speech">杜金房</input>
  </interpretation>
</result>

[INFO] switch_cpp.cpp:1293 Heard! CONFIDENCE = 100
[INFO] switch_cpp.cpp:1293 Heard: 杜金房
```

读者可以看到，实际识别到的内容是以 XML 描述的，其中 confidence 的值表示该语音识别程序有多大的把握保证它听到的是正确的，而真正的检测结果在 input 标签中（即这里的“杜金房”）。接下来，我们只需要用一个 XML 解析器解出相应的内容即可。下面的代码第 16 行将通过 getBody 函数取得 XML，同样第 17 行将识别暂停，第 19 行使用 XML 解析函数 getResults 进而得出结果，并放入 results 变量中，然后打印日志，并在第 23 行用 TTS 进行回放。

```
16             if (obj:getBody()) then
17                 session:execute("detect_speech", "pause")
18                 local speech_ouput = obj:getBody()
19                 results = getResults(obj:getBody())
20                 if(results.score ~= nil) then
21                     log("Heard: CONFIDENCE = " .. results.score .. "\n")
22                     log("Heard: " .. results.text)
23                     s:speak("我听到您说的是" .. results.text .. "\n")
```

这里使用 TTS 把“听到”的内容再跟用户确认一遍，然后执行其他的动作，如根据姓名拨号，调用其他程序的接口控制其他设备等，具体的应用就看读者的想象力了。

本函数其他的代码不是很重要，但出于完整性起见，我们也把它们列在了下面（第 24 ~ 37 行）：

```
24                 else
25                     session:speak("对不起，我听不见你说话")
26                 end
27                 session:sleep(100)
28             end
29             return "break"
30         else
31             session:speak("对不起，我听不见你说话")
32             return "break"
33         end
```

```
34         return "break"
35     end
36     return "break"
37 end
```

在上面的代码中，我们没有考虑识别不成功的情况。在实际的应用中，如果在识别不成功时要重启识别，可以在代码中合适的地方使用 session:execute("detect_speech", "resume") 实现。

当然，上面仅定义了回调函数，实际的动作是在下面执行的。其中，第 43 行安装了回调函数；第 45 行设置了 TTS 的参数；第 46 行开启语音识别，它有三个参数，分别表示使用 pocketsphinx 模块、使用中文（zh）语法文件，以及语法的路径（zh，在这里可以随便填，但不能为空）。第 47 行实际上是播放一个无限长的静音文件并等待你说话。代码如下：

```
38 results = {};
39 speech_detected = false;
40 speech_detected_dest = false;

41 session:sleep(1000);
42 session:answer();
43 session:setInputCallback("onInput");
44 session:sleep(200);
45 session:set_tts_params("tts_commandline", "Ting-Ting");
45 session:speak("请说出一个名字");
46 session:execute("detect_speech", "pocketsphinx zh zh");
47 session:streamFile("silence_stream://0");
```

前面我们讲过，有一些语音识别的参数是在 mod_pocketsphinx 的模块配置文件中配置的，如果需要动态配置，也可以直接使用 detect_speech 的 params 子命令实现。比如，下面代码就设置了另外一个声音识别模型：

```
session:execute("detect_speech","param narrowband-model zh_broadcastnews_
ptm256_8000");
```

另外，我们在此省略了代码中的 XML 解析函数，读者可以参考 http://lua-users.org/wiki/LuaXml 尝试着自己写一个。

（3）测试语音识别

将上面讲的脚本存为 asr.lua 后可以使用如下 Dialplan 测试语音识别：

```
<extension name="ASR demo">
    <condition field="destination_number" expression="^1234$">
        <action application="answer"/>
        <action application="lua" data="asr.lua"/>
    </condition>
</extension>
```

当呼叫 1234 后，便听到语音提示你说出一个名字，此时你可以根据语法文件中的名字说几个来测试一下。测试的输出结果我们上面已经讲过了，在此就不赘述了。

17.8.2 通过商业语音识别软件进行识别

在上面的语音识别例子中，可能是由于笔者配置、调试得不够到位，识别率不是很高。为了给读者提供一个更生动的例子，笔者联系了 Vestec[1]，他们同意提供一个试用的语音识别许可证。Vestec 是加拿大的一家做语音技术的公司，他们的 ASR 产品支持很多种语言，其中一种就是汉语。

Vestec 本身的安装和配置也不是太复杂，但需要花一些时间从各种文档里找到需要的信息，这不是一件轻松的事情。笔者在这里就不详细讲解各种细节了，仅将简单的安装过程罗列一下，以便读者参考。

Vestec 支持多种 Linux 平台，笔者是在 Ubuntu Linux 12.04 上测试的，因此安装了如下的包：

```
# dpkg -i vasre-so_2.10.5.Ubuntu.1204-1_amd64.deb          # 动态库
# dpkg -i vasre-server_2.10.5.Ubuntu.1204-1_amd64.deb      # ASR 服务器
# dpkg -i vasre-rm_2.10.5.Ubuntu.1204-1_amd64.deb          # 资源管理（许可证等）
# dpkg -i vasre-mrcp_2.10.5.Ubuntu.1204-1_amd64.deb        # MRCP 接口
# dpkg -i vasre-lang-zh-CN_2.5.2-1_all.deb                 # 中文包
```

然后，将申请到的许可证文件 license.lic 放到 /opt/Vestec/license 目录中。

我们将使用 MRCP 接口连接语音识别服务。在默认的配置中，MRCP 服务是监听在本地回环接口（127.0.0.1）上，因此我们需要修改 /opt/Vestec/mrcp/conf/mrcpserver.xml 中的 IP 地址，将其中的 auto 修改为实际的 IP 地址。然后启动以下服务：

```
# /etc/init.d/vasre-server start
# /etc/init.d/vasre-rm start
# /etc/init.d/vasre-mrcp start
```

至此 Vestec ASR 服务程序应该就可以正常工作了。在 /opt/Vestec/mrcp/sample-freeswitch 目录中有一些与 FreeSWITCH 集成的示例文件，可以进行测试，不过那些例子都是英文的，我们这里还是讲中文的例子。

Vestec 配置好以后，下面就是配置 FreeSWITCH 端了。首先，我们需要配置 mod_unimrcp 模块。该模块提供 MRCP 协议的支持。MRCP 协议有两个版本，V1 使用 RTSP，V2 使用 SIP，Vestec 和 FreeSWITCH 对两个版本都支持。在这里，我们仅介绍一个 V1 版本的例子。在 FreeSWITCH 安装目录的 conf/mrcp_profiles 中创建 vestec.xml 文件，内容如下（其中 server-ip 为 Vestec ASR MRCP 服务器的 IP，而 rtp-ip 为 FreeSWITCH 的 IP，其他都是默认值）：

```
<include>
  <!-- UniMRCP Server MRCPv1 -->
  <profile name="vestec" version="1">
    <param name="server-ip" value="192.168.7.11"/>
    <param name="server-port" value="1554"/>
    <param name="resource-location" value=""/>
    <param name="speechsynth" value="speechsynthesizer"/>
```

[1] 参见 http://www.vestec.com。

```
    <param name="speechrecog" value="speechrecognizer"/>
    <param name="rtp-ip" value="192.168.7.6"/>
    <param name="rtp-port-min" value="4000"/>
    <param name="rtp-port-max" value="5000"/>
    <!--param name="ptime" value="20"/-->
    <param name="codecs" value="PCMU PCMA L16/96/8000"/>
    <!-- Add any default MRCP params for SPEAK requests here -->
    <synthparams>
    </synthparams>
    <!-- Add any default MRCP params for RECOGNIZE requests here -->
    <recogparams>
      <!--param name="start-input-timers" value="false"/-->
    </recogparams>
  </profile>
</include>
```

mod_unimrcp 模块默认是不编译的，可以到 FreeSWITCH 源代码目录下编译安装它：

```
# make mod_unimrcp-install
```

然后，在 FreeSWITCH 中加载该模块：

```
freeswitch> load mod_unimrcp
```

接下来，还是使用我们在上一节讲的姓名识别的脚本，为了能让 Vestec 识别到我们的关键词，我们按它的语法规则（基于 ABNF[⊖]）来生成一个语法文件 ves-zh.gram 放到 grammar 目录下，内容如下：

```
#ABNF 1.0;
language zh-CN;
tag-format <semantics/1.0-literals>;
root $Name;

$Name =
杜金房      {Seven Du}
    |    孙中山     {Sun Yat-sen}
    |    乔布斯     {Steve Jobs}
    |    令狐冲     {Linghu Chong}
    |    孙悟空     {Sun Wukong};
```

准备好上面这些以后，我们便可以进行测试了。在这里，我们还是使用上一节的 Lua 脚本，只是把 ASR 的调用参数修改为下面一行：

```
session:execute("detect_speech", "unimrcp:vestec ves-zh ves-zh");
```

其中，unimrcp:vestec 指定我们新的 ASR 引擎，ves-zh 则指定我们新的语法文件。接下来，就可以用类似上一节的方法测试了。在笔者的测试过程中，识别效果还是相当不错的。下面是识别结果的 XML 文件，与上一节比起来有一点小小的差别：

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
    <interpretation grammar="session:ves-zh" confidence="99">
```

⊖ Augmented Backus–Naur Form，扩充巴科斯-瑙尔范式，一种基于 BNF 的元语言。参见 http://zh.wikipedia.org/wiki/扩充式巴斯科范式。

```
            <input mode="speech" confidence="99">孙悟空</input>
            <instance confidence="99">
                <Name>Sun Wukong</Name>
            </instance>
        </interpretation>
    </result>
```

另外，笔者也在 Vestec 的日志中发现，它对汉字发音的分析与我们上一节的语言模型有些差异，部分日志内容如下：

```
WARN  Auto-pron generates the pronunciation of 杜金房
INFO  杜金房 : "d uw j ih n f ah ng"
WARN  Auto-pron generates the pronunciation of 孙中山
INFO  孙中山 : "s uh n zh oh ng sh ah n"
WARN  Auto-pron generates the pronunciation of 乔布斯
INFO  乔布斯 : "q ih ao b uw s ir"
WARN  Auto-pron generates the pronunciation of 令狐冲
INFO  令狐冲 : "l ih ng hh uw ch oh ng"
WARN  Auto-pron generates the pronunciation of 孙悟空
INFO  孙悟空 : "s uh n w uw k oh ng"
```

总之，测试结果还是令人满意的。而且，Vestec 对 FreeSWITCH 开源社区也非常友好。如果读者对此有兴趣的话，也可以向他们申请测试的许可证。

除此之外，我们也在本节讲了 mod_unimrcp 的使用和配置，读者可以参照本节的内容配置与其他的 TTS 或 ASR MRCP 服务器对接。

17.9 使用 mod_xml_curl 提供动态用户管理

在本章前面我们讲了一个使用 Lua 来绑定一个回调为 FreeSWITCH 提供 XML Dialplan 的例子，但 Lua 脚本的灵活性还是稍微差一点。因此，这里我们再来看一个用外部的脚本来提供 XML 用户目录的例子。

FreeSWITCH 默认使用静态的 XML 文件配置用户，但如果要动态认证，就需要跟数据库相关联。FreeSWITCH 通过使用 mod_xml_curl 模块完美解决了这个问题。它的实现思路是你自己提供一个 HTTP 服务器，当有用户有注册请求时（或 INVITE 或其他，总之需要 XML 的请求时），FreeSWITCH 向你的 HTTP 服务器发送请求，你查询数据库生成一个标准的 XML 文件，FreeSWITCH 进而通过这一文件所描述的用户信息对用户进行认证。

下面我们先来看一下脚本。该脚本是用 PHP 实现的，PHP 的语法与 C/C++ 以及 Java 都很类似，因此对于不熟悉 PHP 的读者来说，也很容易理解。

首先，在脚本的头部，我们从 $_POST 数组中取得请求参数，如 user（用户）以及 domain 等，然后设置一个 $password 变量，值为 1234。

```
<?php
  $user =  $_POST['user'];
  $domain = $_POST['domain'];
  $password = "1234";
?>
```

作为一个简单的例子，这里我们没有添加多少业务逻辑，下面的代码只是单纯返回了一个 XML 文件，在需要变量的地方便使用 <?php echo $var; ?> 语法直接输出。

```
<document type="freeswitch/xml">
<section name="directory">
  <domain name="<?php echo $domain;?>">
    <params>
        <param name="dial-string" value="{presence_id=${dialed_user}@${dialed_
domain}}${sofia_contact(${dialed_user}@${dialed_domain})}"/>
    </params>
    <groups>
    <group name="default">
      <users>
        <user id="<?php echo $user; ?>">
          <params>
            <param name="password" value="<?php echo $password; ?>"/>
            <param name="vm-password" value="<?php echo $password; ?>"/>
            </params>
          <variables>
            <variable name="toll_allow" value="domestic,international,local"/>
            <variable name="accountcode" value="<?php echo $user; ?>"/>
            <variable name="user_context" value="default"/>
            <variable name="effective_caller_id_name" value="FreeSWITCH-CN"/>
            <variable name="effective_caller_id_number" value="<?php echo $user;?>"/>
            <variable name="outbound_caller_id_name" value="$${outbound_caller_name}"/>
            <variable name="outbound_caller_id_number" value="$${outbound_caller_id}"/>
            <variable name="callgroup" value="default"/>
            <variable name="x-powered-by" value="http://www.freeswitch.org.cn"/>
          </variables>
        </user>
      </users>
    </group>
    </groups>
  </domain>
</section>
</document>
```

通过上述脚本便可以生成 XML 并返回给 FreeSWITCH，然后 FreeSWITCH 再解析该 XML 并进行下一步的认证等操作。因为该脚本根本不查询数据库，也没有任何业务逻辑，任何注册或呼叫请求只要密码是 1234 就都能通过验证，因此我们可以把它称为万能脚本。当然，熟悉 PHP 的读者可以自己尝试添加一些 if…else 类的条件判断以针对不同的用户返回不同的 XML，或者通过查询数据库构造不同的 XML。

接下来，把上述 PHP 文件放到你的 HTTP 服务器上，并确保它能正确执行，然后修改 conf/autoload_configs/xml_curl.conf.xml 配置文件，增加一个 binding，并将 gateway-url 的 value 值设成与你 Web 服务器与 PHP 对应的访问地址，并将 bindings 属性设为 directory。笔者的示例配置如下：

```
<configuration name="xml_curl.conf" description="cURL XML Gateway">
  <bindings>
    <binding name="directory">
      <param name="gateway-url"
```

```
            value="http://localhost/~seven/freeswitch/directory.php"bindings=
"directory"/>
      </binding>
    </bindings>
  </configuration>
```

可以在源代码目录中使用如下命令编译安装该模块：

```
# mod mod_xml_curl-install
```

然后在 FreeSWITCH 控制台上加载该模块：

```
freeswitch> load mod_xml_curl
```

拿起你的 SIP 电话注册试一下吧，别忘了万能密码是 1234。

如果在测试过程中遇到错误，试着看一看出错的内容，一般是 HTTP 服务器无法正常访问造成的。如果没有直接看到错误，也可以在控制台上打开 mod_xml_curl 的调试选项，如：

```
freeswitch> xml_curl debug_on
```

然后，FreeSWITCH 会将每次请求得到的 XML 文件存放到文件名类似 /tmp/xxx.xml 的文件中，检查一下里面的内容是否跟你 HTTP 服务预期的输出是一致的。

17.10 使用 mod_xml_cdr 模块处理话单

与 mod_xml_curl 模块类似，mod_xml_cdr 会在每次生成话单后请求一个 HTTP 服务器，然后 HTTP 服务器就可以进行一些逻辑处理和写入数据库等操作。在 HTTP 服务器端，用户可以使用任何熟悉的语言（如 Java、PHP、Ruby、Python、C# 等）来开发。

在本例中，我们将使用 Ruby 给大家讲解。之所以使用 Ruby，是因为 Ruby 的语法比较易懂，讲起来也比较容易。Ruby 语言之所以能发扬光大倒不是因为它的语法比较好，而是因为有个叫 DHH ㊀的人写了一个叫 Ruby on Rails 的开发框架，利用这个框架能迅速开发 Web 应用，从而使 Ruby 一时风靡 Web 开发领域。不过，我们在这里不使用 Ruby on Rails，而是使用另外一个比较轻量级的框架 Sinatra ㊁。

使用 Sinatra 创建 Web 应用时能更快捷。笔者在这里选用它的另一个原因是，其不像 PHP 那样像个黑洞，它直接有个调试的控制台，可以很容易在日志中打印收到的内容。

Sinatra 的安装很简单，如果你机器上已经有了 Ruby 环境，可以直接通过 gem 来安装，方法如下：

```
# gem install sinatra
```

先来看下面的代码。它的功能很简单，当收到“/cdr”的 POST 请求后，便在控制台上把请求信息中的“cdr”参数打印出来（Sinatra 中所有的请求数据都在 request 对象中），然后返回一个纯文本的回应“CDR Saved”。

㊀ David Heinemeier Hansson，参见 http://en.wikipedia.org/wiki/David_Heinemeier_Hansson。

㊁ 一个用 Ruby 写的轻量级的 Web 开发框架，参见 http://www.sinatrarb.com/。

```
require 'sinatra'

post '/cdr' do
    puts "cdr: #{request["cdr"]}"
    "CDR Saved\n"
end
```

把上面的代码保存到 cdr.rb 文件，然后便可以直接启动一个 Web 服务器了（下面我们称为 CDR 服务器）：

```
$ ruby cdr.rb
INFO  WEBrick 1.3.1
INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.4.2]
== Sinatra/1.4.4 has taken the stage on 4567 for development
INFO  WEBrick::HTTPServer#start: pid=98950 port=4567
```

在日志中我们可以看到它监听了 4567 端口，然后，我们在另外的终端上使用 curl 工具给它发送一个 HTTP POST 请求（FreeSWITCH 默认都是发送 POST 请求），我们就能收到“CDR Saved”回应了。

```
$ curl -XPOST -d cdr=test localhost:4567/cdr
CDR saved
```

同时，在 CDR 服务器的日志中也能看到“cdr: test”输出。接下来，修改 FreeSWITCH 中 mod_xml_cdr 的配置文件，将其中的 url 参数设为指向我们的 CDR 服务器：

```
<param name="url" value="http://localhost:4567/cdr"/>
```

然后在 FreeSWITCH 中重新加载一个 mod_xml_cdr 模块，并打一个电话测试，就能在 CDR 服务器日志中看到一大堆的 XML。前面的一部分如下：

```
cdr: <?xml version="1.0"?>
<cdr core-uuid="bf7f346e-25c8-498b-9a2c-8d0e5f70a266">
  <channel_data>
    <state>CS_REPORTING</state>
    <direction>inbound</direction>
    <state_number>11</state_number>
    <flags>0=1;1=1;36=1;37=1;39=1;42=1;52=1;87=1;111=1;114=1</flags>
    <caps>1=1;2=1;3=1;4=1;5=1;6=1</caps>
  </channel_data>
  <variables>
    <direction>inbound</direction>
    <uuid>6c54299f-ad3b-4d5a-bd32-becbfd290cd2</uuid>
    <session_id>132</session_id>
    <sip_from_user>1006</sip_from_user>
    <sip_from_uri>1006%40192.168.7.6</sip_from_uri>
    <sip_from_host>192.168.7.6</sip_from_host>
```

知道了 XML 的结构，就很容易将其中有用的内容解析出来了，如主被叫号码、呼叫起止时间、通话时长等。

Ruby 中有一个 gem 叫 Nokogiri ⊖，可以用于解析 XML，我们可以用如下命令来安装：

⊖ 一个 Ruby 的 XML 解析器，参见 http://nokogiri.org/。

```
# gem install nokogiri
```

接着，我们使用 Nokigiri 解析 XML[⊖]，并通过 XPath（对应 doc 对象的 xpath 方法）取得相关的内容，并打印出来，代码如下：

```
require 'sinatra'
require 'nokogiri'

post '/cdr' do
    doc  = Nokogiri::XML(request["cdr"])

    caller = doc.xpath("/cdr/variables/user_name//text()")
    dest = doc.xpath("/cdr/callflow/caller_profile/destination_number//text()")
    start_epoch = doc.xpath("/cdr/variables/start_epoch//text()")
    answer_epoch = doc.xpath("/cdr/variables/answer_epoch//text()")
    end_epoch = doc.xpath("/cdr/variables/end_epoch//text()")

    puts "caller: #{caller}"
    puts "dest: #{dest}"
    puts "start_epoch: #{start_epoch}"
    puts "answer_epoch: #{answer_epoch}"
    puts "end_epoch: #{end_epoch}"

    "CDR saved\n"
end
```

上述代码执行后，打一个电话，挂断后日志中的显示结果如下：

```
caller: 1006
dest: 9196
start_epoch: 1385041256
answer_epoch: 1385041256
end_epoch: 1385041257
```

其中，epoch 为 UNIX 时间戳，即 1970 年 1 月 1 日 0 时 0 分 0 秒以来的秒数（不包括闰秒），这样插入数据库等操作就会比较容易实现，当然，为了方便阅读，可以取 stamp 相关的变量，如 start_stamp。

当然，既然能打印出这些值，就很容易将其写入数据库了。在 Sinatra 中进行数据库操作也很简单，不过限于篇幅，我们在此就不多讲了。

17.11 小结

在本章的前面，我们主要以 Lua 脚本作为例子，讲了 IVR、会议控制、外呼、网关连选以及通过 Lua 提供 Dialplan 的内容。通过这些例子，我们不仅复习、巩固了前面学过的知识，同时也穿插补充了一些新的知识，并带领大家初步领略到了 Lua 脚本的魅力。通过对 XML Dialplan 与 Lua Dialplan 以及使用 Lua 绑定提供的 XML Dialplan 的横向和纵向对比，

⊖ XPath 即为 XML 路径语言（XML Path Language），它是一种用来确定 XML 文档中某部分位置的语言，参考 http://zh.wikipedia.org/wiki/XPath 及 http://www.w3school.com.cn/xpath/。

读者应该能更深入理解这里的实现思路、方法和使用技巧。

接着，我们仍然以 Lua 脚本为例给出了语音识别的实例，并对两种语音识别的产品作了简单的对比。虽然由于篇幅限制我们没有写出一个完整的应用，但是抛砖引玉，有了这些基础知识读者就可以进行更深入的研究和学习了。

在后面的部分，我们还通过 PHP 和 Ruby 语言给出使用 mod_xml_curl 和 mod_xml_cdr 的例子。HTTP 技术是很流行的技术，因此，FreeSWITCH 通过使用 HTTP 接口就可以很容易跟其他系统集成。而且，经过这么多年的发展，HTTP 技术已经是非常成熟了，它几乎可以很容易进行横向扩展，无论是从功能还是从性能上讲都不会有瓶颈。因此，我们也可以使用这些技术建立很庞大的 FreeSWITCH 集群。

PHP 是比较流行的 Web 开发语言，Ruby 可能比较小众，但使用 Ruby 进行开发和调试还是非常快速和方便的，即使你在工作环境中不使用 Ruby，笔者也建议你花点时间准备几个 Ruby 脚本，以便快速调试与 FreeSWITCH 的集成。

对编程开发感兴趣的读者可以试着把我们前面的 Lua 脚本扩展一下，增加一些逻辑判断功能、与数据库的交互等。也可以试试用 Ruby 向 FreeSWITCH 提供 XML 数据，用 PHP 解析 CDR 并写入数据库等。通过实践可以帮你把学到的知识掌握得更加牢固。

第 18 章 Event Socket

与 Lua 之类的嵌入式语言不同，通过 Event Socket 方式，可以使用运行在 FreeSWITCH 外部的程序控制 FreeSWITCH。Event Socket 是操控 FreeSWITCH 的“瑞士军刀”。它可以通过 Socket 方式使用 FreeSWITCH 提供的所有的 App 程序和 API 命令。由于绝大多数程序语言都支持 Socket，因而它几乎可以跟任何语言开发的程序通信，也就是说，它几乎可以跟任何系统进行集成。

另外，使用 Event Socket 的一点好处是，它避免了开源许可证的兼容性问题。开源许可证种类繁多，各有特点，如有的对开放源代码要求高些，有的对商业应用限制少些。FreeSWITCH 使用 MPL 1.1 许可证发布，而有些开源许可证（典型的如 GPL）与它是不兼容的，因而，如果 FreeSWITCH 与其他软件或库混合编译就可能遇到许可证兼容方面的问题。通过使用 Event Socket 与其他系统对接，其他任何许可证的软件或系统都是以独立的程序存在的，这样就避免了潜在的许可证兼容问题。

FreeSWITCH 使用 SWIG ㊀来支持多语言。简单来讲，FreeSWITCH 用 C 语言写了一些库函数，通过 SWIG 包封装成其他语言接口。现在已知 FreeSWITCH 支持的语言有 C、Perl、PHP、Python、Ruby、Lua、Java、Tcl，以及由 Managed 支持的 .Net 平台语言如 C#、VB.NET 等。

其实，Event Socket 并没有提供什么新的功能，只是提供了一个开发接口，所有的通道处理及媒体控制都是由 FreeSWITCH 内部提供的 App 和 API 来完成的。

18.1 架构

Event Socket ㊁有两种模式：内连（Inbound）模式和外连（Outbound）模式。注意，这

㊀ Simplified Wrapper and Interface Generator，即简单包装及接口生成器，用于帮助使用 C 或 C++ 语言写的程序生成其他高级语言的接口。参见：http://www.swig.org/。

㊁ 参见 http://wiki.freeswitch.org/wiki/Event_Socket。

里所说的内和外都是针对 FreeSWITCH 而言的。

18.1.1 外连模式

如图 18-1 所示，FreeSWITCH 作为一个 TCP 客户端连接到一个 TCP Server 上。那么这个 TCP Server 是谁呢？它是需要用户自己去实现的，实际上这就是用户需要开发的部分。在这个服务器里，用户可以实现自己的业务逻辑，以及连接数据库获取数据帮助决策等。

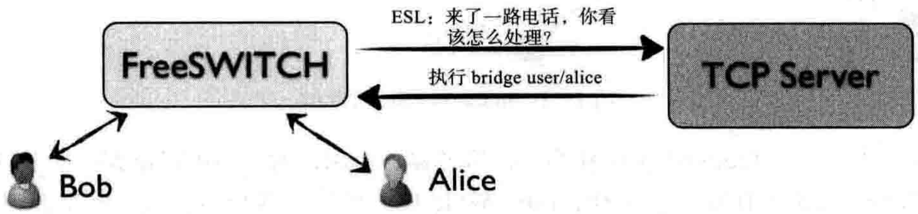


图 18-1 Outbound ESL 示意图

那么，怎么让 FreeSWITCH 去连接你的 TCP Server 呢？回想一下第 7 章。FreeSWITCH 是一个 B2BUA，当 Bob 呼叫 Alice 时，首先电话会到达 FreeSWITCH（通过 SIP），建立一个单腿的 Channel（a-leg），然后电话进入路由状态，FreeSWITCH 查找 Dialplan，然后可以通过以下动作建立一个到 TCP Server 的连接：

```
<action application="socket" data="127.0.0.1:8040"/>
```

到此为止，还是只有一个 Channel。其中 socket 是一个 App，它会先把这个 Channel 置为 Park 状态，然后 FreeSWITCH 作为一个 TCP 客户端连接到 TCP Server 上，把当前呼叫的相关信息告诉它，并询问下一步该怎么做。当然，这里 FreeSWITCH 跟 TCP Server 说的语言称为 ESL，该语言只有它们两个人懂，与 SIP 及 RTP 没有任何关系。也就是说，TCP Server 只是发布控制指令，并不实际处理语音数据。

接下来，TCP Server 收到 FreeSWITCH 的连接请求后，进行决策，如果它认为 Bob 想要呼叫 Alice（根据来话信息和主被叫号码判断），它就给 FreeSWITCH 发一个执行 bridge App 的消息，告诉它应该继续呼叫 Alice（给 Alice 发 SIP INVITE 消息）。

在 Bob 挂机之前，FreeSWITCH 会一直向 TCP Server 汇报 Channel 的相关信息，所以这个 TCP Server 就可掌握这路电话所有的详细信息，也可以在任何时间对它们发号施令。

Bob 挂机后，与 TCP Server 的连接也会断开，并释放资源。读到这里，读者应该明白了。之所以叫做 TCP Server，是因为它应该是一个服务器应用，永远在监听一个端口（在上面的例子中是 8040）等待有人连接。如果需要支持多个连接，这个服务器就应该使用 Socket 的 Select 机制或做成多线程（多进程）的。

我们上面说过，所谓 Inbound 和 Outbound 都是针对 FreeSWITCH 而言的。由于在这种模式下，FreeSWITCH 要向外连接到 TCP Server，因此称为外连模式。

18.1.2 内连模式

内边模式如图 18-2 所示。

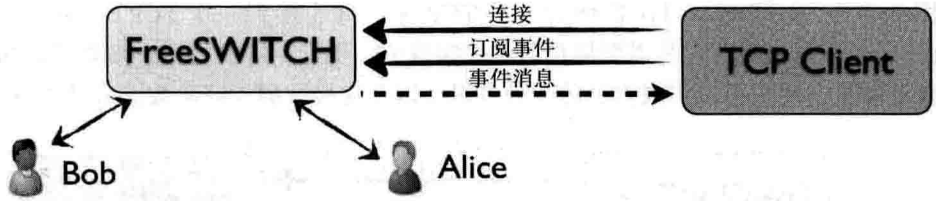


图 18-2 Inbound ESL 示意图

在内连模式下，FreeSWITCH 作为一个服务器，而用户的程序可以作为一个 TCP Client 主动连接到 FreeSWITCH 上。同样，FreeSWITCH 允许多个客户端连接。每个客户端连接上来以后，可以订阅 FreeSWITCH 的一些内部事件。上面我们说过，FreeSWITCH 要通过 EventSocket 向外部发送信息，这些信息就是以事件的形式体现的。同样，在内部好多功能也是事件驱动的。

用户的 TCP Client 收到这些事件后，可以通过执行 App 和 API 来控制 FreeSWITCH 的行为。

对于外连模式来讲，由于 Socket 来自一个 App，而且它所连接的 TCP Server 也像是这个 App 功能的一部分，它们在 Alice 这个 Channel 的内部工作，与之相连的 TCP Server 发布的命令通常也是让 FreeSWITCH 执行一些 App。只是在使用 bridge APP 桥接到 Bob 后这个 Socket Server 又好像是一个中间人或第三者。

对于内连模式，很明显外部的 TCP Client 是一个第三者，它通常不是 Channel 的一部分，而是监听到一个感兴趣的事件以后，通过 API（uuid_ 一族的 API）来对 Channel 进行操作。

18.2 Event Socket 协议

无论是上一节说的哪种模式，FreeSWITCH 都要跟外部的程序“交流思想”，这就要求它们说同一种语言。上面我们为了区分 SIP、RTP 等协议，说这是一种方言。在此，我们姑且称这种方言为 ESL（Event Socket Language，实际上 ESL 是 Event Socket Library，即 Event Socket 库的缩写）。

ESL 协议是纯文本的协议。它的设计思想来自于大家熟悉的 HTTP 协议及 SIP 协议[⊖]。如果对这些协议还不熟悉，可以翻回第 7 章看看。这里再稍微啰嗦一点点——该协议的特点就是大部分消息（一些命令除外）都有一些字段组成的消息头（Header），某些消息有消息体（Body），某些则没有。消息头中有一个名为 Content-Length 的字段，其标志了消息体的长度。消息头和消息体之间用两个回车换行符“\r\n\r\n”隔开。

⊖ 关于这种类型的协议，可以参考一下 http://www.ietf.org/rfc/rfc2822.txt，但 ESL 并不是完全遵循该 RFC。

用户程序（TCP Server /Client）可以使用 sendmsg 发送 App 或直接发送其他 API 命令，此外还有其他相关的认证、订阅事件等操作指令。

下面我们先来看两个外连和内连的例子，然后简单介绍一下 EventSocket 协议的内容。

18.2.1 外连

我们使用如下 Dialplan 进行测试：

```
<extension name="socket">
  <condition field="destination_number" expression="^1234">
    <action application="socket" data="localhost:8040 async full"/>
  </condition>
</extension>
```

当电话呼叫 1234 时，FreeSWITCH 便会使用 Outbound 模式，使用 socket App 启动 Socket 连接。注意这里的两个参数 async 和 full。其中，async 表示异步执行。默认是同步的，比如在同步状态下，如果 FreeSWITCH 正在执行 playback 操作，而 playback 是阻塞的，因而在 playback 完成之前向它发送任何消息都是不起作用的，但异步状态可以进行更灵活的控制。当然，异步状态增加了灵活性的同时也增加了开发的复杂度，读者在实际的开发过程中可以对比一下它们的异同。另一个参数 full 指明可以在外部程序中使用全部的 API，默认只有少量的 API 是有效的。至于哪些 API 跟这个参数相关，也留给读者自行练习。总之一句话，如果你不确定，那么加上 full 是没有错的。

好了，电话来了，由于我们还没有准备好 TCP Server，因此连接会失败，电话就断掉了。

下面我们需要实现一个 TCP Server，在这里我们使用 netcat 这个工具来讲解。

netcat 是一般 Linux 系统自带的一个工具，它可以启动一个 Socket，做服务器或客户端。如果作为客户端，你可以认为它类似于你更熟悉的 telnet 命令。虽然它叫 netcat，但程序的名字是 nc。如果系统默认没有安装，可以尝试使用下列命令来安装：

```
apt-get install netcat        # Ubuntu/Debian
yum install netcat            # CentOS
yum install nc                # 其他?
```

打开一个终端 A，启动一个 Server A，监听 8040 端口（其中 -l 表示监听，即 listen；-k 表示客户端断开后继续保持监听。注意，有些版本的 netcat 参数稍有不同，使用时请查看相关的 man 文档）。

```
nc -l -k localhost 8040
```

打开另一个终端（Terminal）B，启动一个 Client B 连上它：

```
nc localhost 8040
```

然后在这个客户端中打些字，回车，在终端 A 中就应该能看到你输入的文字了。

好了，接下来按 Ctrl+C 退出终端 B，到这里就该让 FreeSWITCH 上场了，即我们用 FreeSWITCH 来替换终端 B。

拿起电话拨打 1234，电话路由到 Socket，FreeSWITCH 就会连到 Server A 上。这时候你听不到任何声音，因为 Channel 处于 Park 状态。但是你在 Server A 里也看不到任何连

接的迹象。不过，如果你打开另一个终端，使用如下命令可以显示 8040 端口已处于连接（ESTABLISHED）状态。

```
$ netstat -an|grep 8040
tcp4       0      0  127.0.0.1.8040        127.0.0.1.60588        ESTABLISHED
tcp4       0      0  127.0.0.1.60588       127.0.0.1.8040         ESTABLISHED
tcp4       0      0  127.0.0.1.8040        *.*                    LISTEN
```

好了，现在回到终端 A，输入 connect 然后打两下回车，奇迹出现了吧？你应该会看到类似下面的输出：

```
Event-Name: CHANNEL_DATA
Core-UUID: 4bfcc9bd-6844-4b45-96a7-4feb2a4f9795
...
```

这便是 FreeSWITCH 发给 Server A 的第一个事件消息，里面包含了该 Channel 所有的信息。下面该 Channel 何去何从就完全看你的了。比如，发送如下消息给它放段音乐（local_stream://moh）：

```
sendmsg
call-command: execute
execute-app-name: playback
execute-app-arg: local_stream://moh
```

建议你直接粘贴上面的命令到 Server A 的窗口里，记得完成后按两下回车。sendmsg 的作用就是发送一个 App 命令（这里是 playback）给 FreeSWITCH，然后 FreeSWITCH 就乖乖地照着做（执行该 App）了。如果你玩够了，就把上面的 playback 换成 hangup 再发一次，电话就挂断了。这些命令就跟直接写到 Dialplan 里一样，不同的是现在是由你来控制，以后你可以用自己的程序控制什么时候该发什么命令。

18.2.2 内连

FreeSWITCH 启动后会启动一个 EventSocket TCP Server ，IP、端口号和密码均可以在 conf/autoload_configs/event_socket.conf.xml 文件里配置。

还记得上面我们在终端 B 里用 nc 做客户端吧？在此，我们还使用 nc 作为客户端，使用以下命令连接 FreeSWITCH：

```
nc localhost 8021
```

连接上以后，你会看到如下消息：

```
Content-Type: auth/request
```

这表示已经连接上 FreeSWITCH 的 Socket 了，并且它告诉你，应该输入密码进行验证。这时我们输入“ auth ClueCon”，记得按两下回车。FreeSWITCH 默认监听在 8021 端口上，默认的密码是 ClueCon，因此我们在上面使用了这些默认值，当然需要的话也可以根据情况在 conf/autoload_configs/event_socket.conf.xml 中修改。

如果一切顺利的话，我们就已经作为一个客户端连接到 FreeSWITCH 上了。可以输入

下列命令试一试（记得每个命令后面都按两下回车）:

```
api version
api status
api sofia status
api uptime
```

你肯定经常在 fs_cli 中使用这些命令，只不过在此我们在每个命令前面多加了个“api”。其实 fs_cli 作为一个客户端也是使用 ESL 与 FreeSWITCH 通信的，只是它帮你做了好多事，你不用手工敲一些协议细节了。但在这里，我们手工输入各种命令，更有助于理解这些细节，例如：

```
api status

Content-Type: api/response
Content-Length: 327

UP 0 years, 0 days, 17 hours, 13 minutes, 19 seconds, 959 milliseconds, 304 microseconds
FreeSWITCH (Version 1.2.11 git b9c9d34 2013-07-20 19:06:40Z) is ready
27 session(s) since startup
0 session(s) - 0 out of max 30 per sec peak 2 (0 last 5min)
1000 session(s) max
min idle cpu 0.00/100.00
Current Stack Size/Max 240K/8192K
```

键入如下命令接收事件：

```
event plain ALL
```

可以订阅所有的事件，当然如果你看不过来可以少订一些。比如，下列命令仅订阅 CHANNEL_CREATE 事件：

```
even plain CHANNEL_CREATE
```

它等效于在 fs_cli 中输入以下命令：

```
/event plain CHANNEL_CREATE
```

18.2.3　Event Socket 命令详解

我们上面举了几个简单的例子，用到了几个命令。实际上，ESL 还提供了更多的命令，用于各种控制。下面列举了大部分在 Event Socket 中可以使用的命令。

1. auth

对于 Inbound 连接来说，auth 是第一个需要发送的命令，用于向 FreeSWITCH 认证，格式如下：

```
auth <password>
```

例如：

```
auth ClueCon
```

实际的密码（这里是 Cluecon）是在 conf/autoload_configs/event_socket.conf.xml 中定义的。

2. api

api 命令上面已经讲过了，它用于执行 FreeSWITCH 的 API，语法如下：

```
api <command> <args>
```

其中，command 和 args 分别是 FreeSWITCH 实际的命令和参数，上面已经举过例子，在此就不多讲了。

3. bgapi

api 命令是阻塞执行的，因此，对于执行时间比较长的 API 命令（如 originate），会有一段时间得不到响应结果，影响实时接收事件。因此，可以使用 bgapi 将这些命令放到后面执行，语法是：

```
bgapi <command> <args>
```

上述命令会立即执行，在后台建立一个任务（Job），并返回一个 Job-UUID，如：

```
Content-Type: command/reply
Reply-Text: +OK Job-UUID: c7709e9c-1517-11dc-842a-d3a3942d3d63
```

当真正需要执行的 API 命令返回后，FreeSWITCH 会产生一个 BACKGROUND_JOB 事件，该事件带了原先的 Job-UUID 以及命令执行的结果。因而客户端可以通过匹配 Job-UUID 知道上一次命令的执行结果。

另外，FreeSWITCH 也允许客户端提供自己的 Job-UUID，这样匹配起来就更容易一些（但客户端需要保证产生的 Job-UUID 要全局唯一），命令格式是：

```
bgapi <command> <args>
Job-UUID: <uuid>
```

如下面的例子：

```
event plain BACKGROUND_JOB        （订阅 BACKGROUND_JOB 事件）

Content-Type: command/reply       （上述指令的返回结果）
Reply-Text: +OK event listener enabled plain

bgapi status                      （使用 bgapi 执行 status 命令）
Job-UUID: My-UUID                 （指定 Job-UUID）

Content-Type: command/reply       （上述指令的返回结果）
Reply-Text: +OK Job-UUID: My-UUID
Job-UUID: My-UUID

Content-Length: 852               （在此我们收到一条消息，长度是 852 字节）
Content-Type: text/event-plain    （消息类型是 text/event-plain）

Event-Name: BACKGROUND_JOB        （消息中的内容是一个事件——Event，它的名字是 BACKGROUND_JOB）
...                               省略一些事件的消息头）
Event-Sequence: 9308
Job-UUID: My-UUID                 （从该头域中可以看到 Job-UUID 与我们在上述 bgapi 中指定的匹配）
Job-Command: status               （该事件的正文本部分就包含了 status 命令的输出）
Content-Length: 325               （事件正文部分的长度，接下来是事件正文，即 status 命令的输出）
```

```
UP 0 years, 0 days, 17 hours, 1 minute, 27 seconds, 717 milliseconds, 221 microseconds
FreeSWITCH (Version 1.2.11 git b9c9d34 2013-07-20 19:06:40Z) is ready
27 session(s) since startup
0 session(s) - 0 out of max 30 per sec peak 2 (0 last 5min)
1000 session(s) max
min idle cpu 0.00/100.00
Current Stack Size/Max 240K/8192K
```

4. linger 和 nolinger

在外连模式下，当一个 Channel 挂断时，FreeSWITCH 会断开与 TCP Server 的 Socket 的连接。这时，可能还有与该 Channel 相关的事件没有来得及发送到 TCP Server 上，因而会“丢失”事件。为避免这种情况发生，TCP Server 可以明确告诉 FreeSWITCH 希望它能在断开之前“逗留”(linger) 一段时间，以等待把所有事件都发完。格式如下：

```
linger <seconds>
```

例如：

```
linger 10
```

如果开启了 linger 又后悔了，可以再用 nolinger 命令撤销，该命令没有参数。

5. event

event 用于订阅事件。让 FreeSWITCH 把相关的事件发送过来。格式是：

```
event [type] <events>
```

其中，type（即事件类型）有 plain、json 和 xml 三种，默认为 plain，即纯文本；events 参数可以指定事件的名字，或者使用 ALL 表示订阅全部事件。订阅全部事件的命令如下：

```
event plain ALL
```

仅订阅部分事件，事件名字之间以空格隔开：

```
event plain CHANNEL_CREATE CHANNEL_ANSWER CHANNEL_HANGUP_COMPLETE
```

要想订阅 CUSTOM 事件该怎么做呢？ CUSTOM 事件即自定义事件，它是一类特殊的事件，它主要是为了扩充 FreeSWITCH 的事件类型，它具体的类型是在 Subclass 中指定的，如下面的指令指定订阅 Subclass 为“sofia::register”的事件：

```
event plain CUSTOM sofia::register
```

可以一次订阅多个 CUSTOM 事件，如：

```
event plain CUSTOM sofia::register sofia::unregister sofia::expire
```

也可以使用多个 event 命令混合订阅，如：

```
event plain CHANNEL_ANSWER CHANNEL_HANGUP
event plain CHANNEL_BRIDGE CUSTOM sofia::register sofia::unregister
```

但参数中一旦出现了 CUSTOM，后面就不能有普通的事件类型了。如下面的订阅方法是达不到预期的效果的（FreeSWITCH 会把 CHANNEL_ANSWER 当成 CUSTOM 事件的 Subclass 对待，因而不是你想要的）：

```
event plain CHANNEL_CREATE CUSTOM sofia::register CHANNEL_ANSER
```

另外，CUSTOM 事件必须逐一明确订阅，如下面这种订阅形式是不对的：

```
event plain CUSTOM ALL
```

其他的例子还有：

```
event json CHANNEL_CREATE
event xml CHANNEL_CREATE
event plain DTMF
event plain ALL
```

最后，值得一提的是，HEARTBEAT 是一个特殊的事件，它每 20 秒就产生一次，用于汇报 FreeSWITCH 的当前状态。有时候可以用它做心跳，如果超过 20 秒没收到事件，就可以认为网络或 FreeSWITCH 异常。下面是 HEARTBEAT 事件的三种不同输出格式：

json 格式：

```
Content-Length: 939
Content-Type: text/event-json

{
    "Event-Name":   "HEARTBEAT",
    ...
    "Event-Info":   "System Ready",
    "Up-Time":  "0 years, 0 days, 22 hours, 19 minutes, 14 seconds, ...
    ...
}
```

XML 格式：

```
Content-Length: 1432
Content-Type: text/event-xml

<event>
  <headers>
    <Event-Name>HEARTBEAT</Event-Name>
    ...
    <Event-Info>System%20Ready</Event-Info>
      <Up-Time>0%20years,%200%20days,%2022%20hours,%2019%20minutes,%2034%20
seconds,...</Up-Time>
    ...
  </headers>
</event>
```

PLAIN（纯文本）格式：

```
Content-Length: 848
Content-Type: text/event-plain

Event-Name: HEARTBEAT
...
Event-Info: System%20Ready
Up-Time: 0%20years,%200%20days,%2022%20hours,%2019%20minutes,%2054%20seconds,...
...
```

6. myevents

myevents 是 event 的一种特殊情况，它主要用于 Outbound 模式中。在 Outbound 模式中，对于每一个呼叫（对应一个 Channel），FreeSWITCH 都会向外部的 TCP Server 请求以建立一个新的连接。外部的 TCP Server 就可以通过 myevents 订阅与该 Channel（UUID）相关的所有事件。使用的格式为：

```
myevents <type><UUID>
```

当然，myevents 也支持 json 及 XML 形式，如：

```
myevents
myevents json
myevents xml
```

当然，在 Inbound 模式中也可以调用 myevents，这样它看起来类似于一个 Outbound 模式的连接，但它要指定 UUID，原因是显而易见的，如：

```
myevents 289fe829-af62-47be-9a59-7519a77d0d40
```

7. divert_events

还有一类特殊的事件，它们是作为 InputCallback 产生的。那么什么时候会产生 InputCallback 呢？回忆一下 16.2.3 节，当 Channel 上通过 setInputCallback() 函数安装了相关的回调函数并遇到某些事件，如收到用户按键（DTMF）或收到语音识别的结果（DETECTED_SPEECH），就会产生 InputCallback 这样的事件，并回调指定的回调函数，但这些 InputCallback 默认只能在嵌入式脚本的回调函数中捕获。通过使用 divert_events，就能将这些事件转发到 Event Socket 连接上来，进而在通过 Event Socket 连接的外部程序中也能收到相关事件。使用格式是：

```
divert_events on            # 开启
divert_events off           # 关闭
```

7. filter

filter 用于安装一个过滤器。这里的过滤不是“滤出”，而是“滤入”，即把符合过滤条件的过滤进来，也就是我们要收到它们。可以同时使用多个过滤器，使用格式是：

```
filter <EventHeader> <ValueToFilter>
```

例如，下面的例子与 myevent <uuid> 的作用是相同的：

```
event plain all
filter Unique-ID <uuid>
```

又如，下面的例子会订阅所有事件，但只接收匹配主叫号码是 1001 的事件：

```
event plain all
event filter Caller-Caller-ID-Name 1001
```

为了理解滤入的概念，可以看下面的例子以加深印象，它可以接收 3 个 Channel 事件：

```
event plain all
filter Unique-id uuid1
```

```
filter Unique-ID uuid2
filter Unique-ID uuid3
```

如果过滤器写错了，或不想使用某些过滤器了，则可以将其取消掉，如：

```
filter delete                    # 取消所有的过滤器
filter delete Unique-ID uuid2    # 仅取消与 uuid2 相关的过滤器
```

8. nixevent 与 noevent

nixevent 是 event 的反义词，与 event 的语法一样，只是取消某些已订阅的事件。如：

```
nixevent CHANNEL_CREATE
nixevent all
```

另外，还有一个 noevent 命令用于简单取消使用 event 订阅的所有事件，相当于“nixevent all”。

9. log

跟使用 event 订阅事件类似，log 用来订阅日志，它的使用格式是：

```
log <level>
```

其中 level 级别我们在 6.1.4 节已经讲过了，如：

```
log info
log 6
```

下面是一个完整的例子。这个例是在开启了 info 级别的日志以后打了一个电话，此时会收到很多日志信息，下面是两条信息：

```
$ nc 127.0.0.1 8022
Content-Type: auth/request

auth ClueCon

Content-Type: command/reply
Reply-Text: +OK accepted

log info

Content-Type: command/reply
Reply-Text: +OK log level info [6]

Content-Type: log/data
Content-Length: 141
Log-Level: 5
Text-Channel: 3
Log-File: switch_channel.c
Log-Func: switch_channel_set_name
Log-Line: 1030
User-Data: dfe873c8-f418-4807-8f45-0c87402fdebd

2013-07-23 00:22:17.293556 [NOTICE] switch_channel.c:1030 New Channel sofia/
internal/1002@192.168.7.5 [dfe873c8-f418-4807-8f45-0c87402fdebd]
Content-Type: log/data
Content-Length: 103
```

```
Log-Level: 6
Text-Channel: 3
Log-File: mod_dialplan_xml.c
Log-Func: dialplan_hunt
Log-Line: 558
User-Data: dfe873c8-f418-4807-8f45-0c87402fdebd

2013-07-23 00:22:17.353573 [INFO] mod_dialplan_xml.c:558 Processing Seven
<1002>->s in context default
```

10. nolog

log 的反义词，关闭使用 log 命令订阅的日志，具体的使用方法很简单，这里不再介绍。

11. exit

告诉 FreeSWITCH 关闭 Socket 连接。FreeSWITCH 收到该命令后会主动关闭 Socket 连接。

12. sendevent

通过 sendevent 可以向 FreeSWITCH 的事件系统发送事件。使用格式是：

```
sendevent <event-name>
```

比如，你可以发送一个 NOTIFY 消息：

```
sendevent NOTIFY
profile: internal
event-string: check-sync
user: 1002
host: 192.168.7.5
content-type: application/xml
content-length: 29

<xml>FreeSWITCH IS COOL</xml>
```

FreeSWITCH 收到 NOTIFY 消息后，将启用内部处理机制，最后它可能会生成一个 SIP NOTIFY 消息，如：

```
NOTIFY sip:1002@192.168.7.5:32278;rinstance=3db08ce44e5166a4 SIP/2.0
Via: SIP/2.0/UDP 192.168.7.5;rport;branch=z9hG4bKXDtD32g0N9atg
Max-Forwards: 70
From: <sip:1002@192.168.7.5>;tag=ZF9SFyaUHeZ4p
To: <sip:1002@192.168.7.5>
...
Event: check-sync
Subscription-State: terminated;reason=noresource
Content-Type: application/xml
Content-Length: 29

<xml>FreeSWITCH IS COOL</xml>
```

当然，也可以使用它发送 MESSAGE 消息，如：

```
sendevent SEND_MESSAGE
profile: internal
```

```
user: 1002
host: 192.168.7.5
content-type: text/plain
content-length: 10

Hello 1002
```

上述命令将会产生如下的 SIP 消息：

```
send 623 bytes to udp/[192.168.7.5]:32278 at 16:01:23.686473:
------------------------------------------------------------------------
MESSAGE sip:1002@192.168.7.5:32278;rinstance=3db08ce44e5166a4 SIP/2.0
Via: SIP/2.0/UDP 192.168.7.5;rport;branch=z9hG4bK085Q8K3aD4DjK
Max-Forwards: 70
From: <sip:1002@192.168.7.5>;tag=11UBKmc2B0Bae
To: <sip:1002@192.168.7.5>
...
Content-Type: text/plain
Content-Length: 10

Hello 1002
```

笔者使用 Bria 注册的 SIP 客户端收到消息的界面如图 18-3 所示。

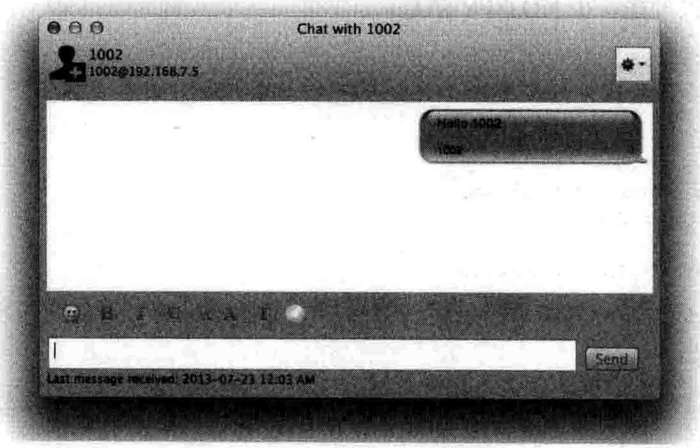


图 18-3 Bria 收到 SIP MESSAGE 后的显示

当然，在实际使用中更多的是产生 CUSTOM 的事件，可以自定义一些事件类型。使用这种方式甚至可以把 FreeSWITCH 当成一个消息队列（Message Queue）来用，如你可以启动一个客户端订阅以下消息：

```
event plain CUSTOM freeswitch:book
```

我们可以在另外的客户端上发送一条消息，如：

```
sendevent CUSTOM
Event-Subclass: freeswitch::book
content-type: text/plain
```

```
content-length: 44

This Message comes from the FreeSWITCH Book!
```

我们可以在订阅该消息的客户端上收到如下信息：

```
Content-Length: 688
Content-Type: text/event-plain

Event-Subclass: freeswitch%3A%3Abook
Command: sendevent%20CUSTOM
...
content-type: text/plain
Content-Length: 44

This Message comes from the FreeSWITCH Book!
```

18.3 Event Socket 库

事情永远都不像看起来那么简单。如果所有协议都需要手工敲的话，那电话一多你就忙不过来了。不过，这些早就有人帮你做好了，这就是我们下面要讲的 ESL（Event Socket Library，Event Socket 库）。ESL 提供了一些库函数，通过这些库函数可以很方便地使用 ESL 协议与 FreeSWITCH 交互，进而控制 FreeSWITCH 的各种功能。

在上面的实验中，你应该已注意到原始的 ESL 协议中有很多类似下面的输出：

```
Content-Type: command/reply
```

这些是针对你输入命令的响应，如果同时你订阅了事件的话还能收到很多事件。再如果你有很多通话，而同时你又要不停地控制它们，这时会不会很凌乱？会不会产生竞争条件（Race Condition）？答案是不会，因为 ESL 都帮你做好了，它将相关的功能模块都封装成了函数，而且有各种语言的版本。当然，对于不支持的语言，就需要自己写了。比如，笔者在几年前学习 Flex 时就曾自己写了一个 Flex 版本的 ESL 库[⊖]。

虽然是实际应用中一般不需要知道上面的 ESL 协议的具体细节，但笔者相信了解了这些协议的具体内容，有助于理解 ESL。

18.3.1 Event Socket 示例

言归正传，其实学习编程序最好的办法就是看别人写的代码。因此，下面我们先来看几个使用 ESL 的例子，然后再对 ESL 中提供的函数进行说明。

1. Ruby 客户端

下面是一个使用 Ruby 语言通过 ESL 控制 FreeSWITCH 的例子。不管你是否熟悉 Ruby，以下脚本都是很好懂的，脚本内容如下：

```
require 'ESL'
```

⊖ 参见：http://wiki.freeswitch.org/wiki/FsAir。

```
con = ESL::ESLconnection.new('127.0.0.1', '8021', 'ClueCon')
esl = con.sendRecv('api sofia status')
puts esl.getBody
```

上述脚本只有短短的 4 行代码，首先它加载了 ESL 库，然后连接到 FreeSWITCH，接着执行 sofia status 命令，最后将结果输出到控制台。

2. C 语言客户端示例

在源代码目录的 libs/esl/ 目录下有个 testclient.c 的 ESL 客户端例子，编译运行它后会使用 Inbound 模式连接到 FreeSWITCH。对于下面的 testclient.c 的内容，为了讲解方便我们进行了简单的修改并增加了中文注释：

```
int main(void)
{
    // 初始化一个 handle，用于标志到 FreeSWITCH 的 Socket 连接
    esl_handle_t handle = {{ 0 }};

    // 连接服务器，如果成功 handle 就代表连接成功了
    esl_connect(&handle, "localhost", 8021, NULL, "ClueCon");

    // 发送一个命令，并接收返回值
    esl_send_recv(&handle, "api status\n\n");

    // last_sr_event 应该是 last server response event，即针对上面命令的响应
    // 如果在此之前收到事件（在本例中不会，因为没有订阅），事件会存到一个队列里，不会发生竞争条件

    if (handle.last_sr_event&& handle.last_sr_event->body) {
        // 打印返回结果
        printf("%s\n", handle.last_sr_event->body);
    } else {
        // 对于 api 和 bgapi 来说（上面已经将命令写死了），应该不会运行到这里，但其他命令可能
        // 会到这里
        printf("%s\n", handle.last_sr_reply);
    }

    // 断开连接
    esl_disconnect(&handle);
    return 0;
}
```

可以看出，该程序很简单，它运行之后向 FreeSWITCH 建立一个连接，运行一个 API 命令，然后输出命令的执行结果。

3. C 语言服务器示例

在 testclient.c 所在目录中有一个 testserver.c，其运行于 Outbound 模式，是多线程的，每次有电话进来时，通过 Dialplan 路由到 socket App。该 App 便会从 FreeSWITCH 中向 testserver 发起一个 TCP 连接。

我们先从 main 函数开始看。首先它调用 esl_global_set_default_logger 函数设置日志级别（代表 DEBUG 级别，即最大级别，这样能看到详细的日志，包括所有协议的细节）。然后它通过 esl_listen_threaded 启动一个 Socket 监听本地回环地址（localhost）的 8040 端

口。如果有连接（从 FreeSWITCH）到来，它便回调 mycallback 函数为该连接服务。其中，NULL 是一个空指针，该参数的位置是一个无类型（void *）的指针，即允许你传入任何类型的指针，该指针将作为 mycallback 函数的参数（user_data）在回调中携带[⊖]。

```
int main(void)
{
    esl_global_set_default_logger(7);
    esl_listen_threaded("localhost", 8040, mycallback, NULL, 100000);
    return 0;
}
```

下面我们介绍回调函数，该函数比较长，因此为了描述方便我们加了行号。注意，这里的行号仅为描述方便，与原始文件中的行号不同。

首先，第 1 行为函数的定义，它有三个参数，分别是服务器端的 Socket 标志、客户端的 Socket 标志、一个连接地址，以及一个用户私有数据。第 3 行初始化了一个句柄，用于标志本次 ESL 连接，其他几个变量的意义我们后面再说。具体代码如下：

```
    01 static void mycallback(esl_socket_t server_sock, esl_socket_t client_sock,
struct sockaddr_in *addr, void *user_data)
    02 {
    03     esl_handle_t handle = {{ 0 }}; // 初始化一个句柄，用于标志一个 ESL 连接
    04     int done = 0;
    05     esl_status_t status;
    06     time_t exp = 0;
```

第 7 行，将用于标志 ESL 连接的 handle 与传入的客户端的 Socket 绑定，这样就可以与客户端之间相互收发消息了；第 8 行，打印一条日志；第 11 行，添加一个过滤器（filter），只收取与本次连接的 Channel 相关的事件（与本次连接的 Channel UUID 相同的事件）；第 12 ～ 15 行，订阅各种类型的事件，至此我们应该能收到跟本 Channel 相关的事件了。具体代码如下：

```
    07    esl_attach_handle(&handle, client_sock, addr); // 将 handle 与 socket 绑定
    08    esl_log(ESL_LOG_INFO, "Connected! %d\n", handle.sock);
    11    esl_filter(&Handle, "unique-id", esl_event_get_header(handle.info_event,
                   caller-unique-id"));
    12    esl_events(&handle, ESL_EVENT_TYPE_PLAIN, "SESSION_HEARTBEAT CHANNEL_NSWER"
              " CHANNEL_ORIGINATE CHANNEL_PROGRESS CHANNEL_HANGUP "
    14        " CHANNEL_BRIDGE CHANNEL_UNBRIDGE CHANNEL_OUTGOING CHANNEL_EXECUTE"
    15        " CHANNEL_EXECUTE_COMPLETE DTMF CUSTOM conference::maintenance");
```

第 19 行，发送一个 linger 命令开启逗留模式。如果不开启该模式，则主叫挂机后，FreeSWITCH 会立即断开它主动建立的 Socket，这样会造成一些后续的事件收不到，如 CHANNEL_HANGUP_COMPLETE、CHANNEL_DESTROY 等。逗留模式的目的就是告诉 FreeSWITCH 晚一些断开这个 Socket。后面我们还会讲到 linger 的用法。第 20 行，执行 answer App 对来话进行应答（跟在 Dialplan 中类似），然后，在第 21 行将来话送入一个会

⊖ 也就是它允许你从 main 函数向回调函数传递私有数据。注意，该参数直到 1.4 版（及 Git master 版）才有，在 1.2 版中无此参数。

议。具体代码如下：

```
19    esl_send_recv(&handle, "linger");
20    esl_execute(&handle, "answer", NULL, NULL);
21    esl_execute(&handle, "conference", "3000@default", NULL);
```

接下来是一个无限循环，用于不断地接收事件。在第 22 行，我们用的接收事件函数 esl_recv_timd 是非阻塞的，如果在 1 秒内没有收到任何事件，它就会返回，然后程序进入循环体。在第 23 行，检测 done 变量，它是个结束标志，用于结束循环，在满足一定的条件下，将在第 32 行对其进行赋值。在 24 行，判断，即如果当前的时间超过 exp 变量指定的值，则退出循环（关于退出条件我们后面再讲）。一般情况下，程序会执行到第 25 行，如果成功收到一条消息，则在第 26 行取出消息的类型。在此，它忽略所有的消息，直到在满足第 27 行的条件（挂机时将收到该消息）。如果收到挂机消息，则第 28 行将取出 FreeSWITCH 发过来的关于挂机的相关信息，并在第 29 行打印一条日志。具体代码如下：

```
22    while((status = esl_recv_timed(&handle, 1000)) != ESL_FAIL) {
23        if (done) {
24            if (time(NULL) >= exp) { break; }
25        } else if (status == ESL_SUCCESS) {
26            const char *type = esl_event_get_header(handle.last_event, content-type");
27            if (type && !strcasecmp(type, "text/disconnect-notice")) {
28                const char *dispo = esl_event_get_header(handle.last_event,
                  content-disposition");
29                esl_log(ESL_LOG_INFO, "Got a disconnection notice
                  dispostion: %s]\n", dispo ? dispo : "");
```

由于我们使用了逗留模式，因此 FreeSWITCH 给我们返回的消息中将包含 linger 字符（或者从打印出的日志中看到），因此，我们设置 done 变量为 1，并于第 35 行将 exp 变量的值设为 5 秒，也就是说，在 5 秒之后循环中将匹配第 23 ~ 24 行的条件。具体代码如下：

```
31                if (!strcmp(dispo, "linger")) {
32                    done = 1;
33                    // 挂机后，可能还有后续的事件，我们多等几秒钟
34                    esl_log(ESL_LOG_INFO, "Waiting 5 seconds for any remaining
                              events.\n");
35                    exp = time(NULL) + 5;
36                }
37            }
38        }
39    }
```

上面的消息接收循环完毕后将打印一条日志（第 40 行），并断开连接（第 41 行，这时很可能连接已经断开了，这样做只是为了保险）。

```
40    esl_log(ESL_LOG_INFO, "Disconnected! %d\n", handle.sock);
41    esl_disconnect(&handle);
42 }
```

总之，该程序也很简单，它启动后监听一个端口，当 FreeSWITCH 中有来话路由到它时便启动一个新线程为新的 Channel 进行服务（具体的服务就是应答），并将来话送入一个会议。由于它是多线程的，因而可以同时为很多来话服务。

4. 使用 ESL 发送 SIP MESSAGE 消息

Wiki[⊖]上有一个使用 Perl 发送 MESSAGE 的应用，这里我们用 C 改写一下（在 testclient 基础上修改的）。

首先，前三行装入了必要的头文件。程序比较简单，只有一个 main 函数（第 5 行），并于第 7 ~ 9 行初始化了一些变量。程序内容如下。

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <esl.h>
04
05 int main(void)
06 {
07     esl_handle_t handle = {{ 0 }};
08     struct esl_event *event;
09     struct esl_event_header header;
```

第 10 行初始化一个 CUSTOM 类型的事件，它的子类型（Subclass）是 SMS::SEND_MESSAGE。第 11 ~ 17 行则给该事件增加各种必要的参数。关于这些参数的具体含义我们就不多讲了，只需要知道，FreeSWITCH 收到包含这些参数的事件后，将会给 1000@192.168.0.7 用户发送一个文本消息，消息内容是在第 17 行定义的。具体代码如下：

```
10     esl_event_create_subclass(&event,  ESL_EVENT_CUSTOM, "SMS::SEND_MESSAGE");
11     esl_event_add_header_string(event, ESL_STACK_BOTTOM, "to", "1000@192.168.0.7");
12     esl_event_add_header_string(event, ESL_STACK_BOTTOM, "from", "seven@192.168.0.7");
14     esl_event_add_header_string(event, ESL_STACK_BOTTOM, "sip_profile", "internal");
15     esl_event_add_header_string(event, ESL_STACK_BOTTOM, "dest_proto", "sip");
16     esl_event_add_header_string(event, ESL_STACK_BOTTOM, "type", "text/plain");
17     esl_event_add_body(event, "Hello");
```

第 18 行，建立到 FreeSWITCH 的连接。第 19 行，发一个命令试试连接是否正常。第 20 行，如果连接正常的话，打印一下上述命令的输出结果（第 21 行），并于第 23 行将刚才构造的事件发送出去。当然，要记得在第 24 行释放事件占用的内存。当然，如果第 19 行执行失败的话，程序会执行到第 29 行，打印相关信息。具体代码如下：

```
18     esl_connect(&handle, "localhost", 8021, NULL, "ClueCon");
19     esl_send_recv(&handle, "api version\n\n");
20     if (handle.last_sr_event&& handle.last_sr_event->body) {
21         printf("%s\n", handle.last_sr_event->body);
22         printf("sending event....\n");
23         esl_sendevent(&handle, event);
24         esl_event_destroy(&event);
28     } else {
29         printf("%s\n", handle.last_sr_reply);
30     }
```

最后，断开连接，程序正常返回（第 31 ~ 32 行）。具体代码如下：

```
31     esl_disconnect(&handle);
32     return 0;
33 }
```

⊖ 参见：http://wiki.freeswitch.org/wiki/Mod_sms#Sending_a_Message_via_ESL。

执行以上程序后，在 1000 这个 SIP 电话上将会收到一个 Hello 消息（如果该用户正常注册的话）。

18.3.2 ESL 函数说明

通过上面的几个简单的例子，相信读者已经对 ESL 有了一个基本的了解，下面我们就来详细看一下 ESL 里涉及的基本概念以及它提供的库函数。

18.2.3 节中所讲的 Event Socket 命令对于我们理解各种基本概念是非常有用的。除此之外，FreeSWITCH 也通过 ESL 库包装了大量的易于使用的函数，我们在 18.3.1 节也已经领略到一部分函数的风采了。

ESL 本身与 FreeSWITCH 没有任何依赖关系，可以单独编译和使用。它在底层是使用 C 语言实现的，并通过 swig 包装成了其他程序语言惯用的格式。Wiki 上的例子大部分都是以 Perl 和 PHP 语言编写的，为了方便读者比较，本节也给出了 C 语言对应的形式，以方便使用 C 语言开发的读者参考。即使不使用 C 语言而使用其他语言开发，对照原代码也便于发现在 Wiki 上找不到的函数原型及数据结构。

1. ESL 对象

ESL 对象是一个核心 ESL 的对象，它用于一些全局的设置。你可以使用如下函数设置打印日志的级别：

```
eslSetLogLevel($loglevel)
```

它的 C 语言定义如下：

```
ESL_DECLARE(void) esl_global_set_default_logger(int level);
```

这里的 level 是整数值，对应关系如下：

- 0 为 EMERG；
- 1 为 ALERT；
- 2 为 CRIT；
- 3 为 ERROR；
- 4 为 WARNING；
- 5 为 NOTICE；
- 6 为 INFO；
- 7 为 DEBUG。

2. ESL Event 对象

ESL Event 是一个事件对象，在 C 语言中的定义如下（在 esl_event.h 中定义）：

```
/*! \brief Representation of an event */
struct esl_event {
    esl_event_types_t event_id;         /*! 事件 ID （描述符） */
    esl_priority_t priority;            /*! 事件优先级 */
    char *owner;                        /*! 事件所有者 */
    char *subclass_name;                /*! 事件的子类型(Subclass) */
```

```
        esl_event_header_t *headers;          /*! 事件的头域，指向一个链表 */
        esl_event_header_t *last_header;      /*! 指向事件的最后一个头域 */
        char *body;                           /*! 事件的正文（内容）*/
        void *bind_user_data;                 /*! 事件绑定者提供的用户自定义数据 */
        void *event_user_data;                /*! 事件发送者绑定的用户自定义数据 */
        unsigned long key;                    /*! 唯一的键值 */
     struct esl_event next;                   /*! 下一个事件的指针 /
    int flags;                                /*! 标志 */
    };
```

当从 FreeSWITCH 中收到一个事件后，你就得到一个事件对象。ESL 定义了一些函数用于从该对象中获取信息或构造新的对象。下面我们来简单介绍。

- serialize([$format])。它可以将事件序列化成可读的形式，以冒号分隔的"名称 : 值"的形式给出。C 语言定义如下：

```
    ESL_DECLARE(esl_status_t) esl_event_serialize(esl_event_t *event, char **str,
esl_bool_t encode);
```

- setPriority([$number])。设置事件的级别。C 语言定义如下：

```
    ESL_DECLARE(esl_status_t) esl_event_set_priority(esl_event_t *event,
esl_priority_t priority);
```

- getHeader($header_name)。从事件中获取头域的值。其 C 语言定义如下：

```
    ESL_DECLARE(char *) esl_event_get_header_idx(esl_event_t *event, const
char *header_name, int idx);
    #define esl_event_get_header(_e, _h) esl_event_get_header_idx(_e, _h, -1)
```

- getBody()。从事件中获取正文：

```
    ESL_DECLARE(char *) esl_event_get_body(esl_event_t *event);
```

- getType()。取得事件的类型。在 C 语言中可以使用下列方式实现：

```
    esl_event_get_header(event, "content-type");
```

- addBody($value)。向事件中增加正文，可以调用多次。C 语言定义如下：

```
    ESL_DECLARE(esl_status_t) esl_event_add_body(esl_event_t *event, const
char *fmt, ...);
```

- setBody($value)。设置事件的正文，可以多次调用，但后者将覆盖前者。C 语言定义如下：

```
    ESL_DECLARE(esl_status_t) esl_event_set_body(esl_event_t *event, const char *body);
```

- addHeader($header_name, $value)。向事件中增加一个头域。C 语言定义如下：

```
    ESL_DECLARE(esl_status_t) esl_event_add_header_string(esl_event_t *event,
esl_stack_t stack,
    const char *header_name, const char *data);
```

C 语言中还有另一种形式，即使用类似 printf 的方式设置字符串的值：

```
    ESL_DECLARE(esl_status_t) esl_event_add_header(esl_event_t *event,
        esl_stack_t stack, const char *header_name, const char *fmt, ...);
```

❑ delHeader($header_name)。从 Event 中删除头域。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_event_del_header_val(esl_event_t *event,
    const char *header_name, const char *var);
#define esl_event_del_header(_e, _h) esl_event_del_header_val(_e, _h, NULL)
```

❑ firstHeader()。将指针指向 Event 的第一个头域，并返回它的 Key 值。C 语言中没有明确的定义，靠使用 ESL Event 的 headers 结构体成员实现。

❑ nextHeader()。移动指针指向下一个 header，在调用该函数前必须先调用 firstHeader()，同样在 C 语言中没有明确定义，靠访问 esl_event_header 结构体的 next 成员实现。

3. ESLconnection 对象

ESLConnection 对象维护与 FreeSWITCH 之间的连接，以发送命令并进行事件处理。在 C 语言中使用如下定义（在 esl.h 中）：

```
typedef struct {
    struct sockaddr_storage sockaddr;
    struct hostent hostent;
    char hostbuf[256];
    esl_socket_t sock;
    /*! In case of socket error, this will hold the error description as reported
    by the OS */
    char err[256];
    /*! The error number reported by the OS */
    int errnum;
    /*! The inner contents received by the socket. Used only internally. */
    esl_buffer_t *packet_buf;
    char socket_buf[65536];
    /*! Last command reply */
    char last_reply[1024];
    /*! Las command reply when called with esl_send_recv */
    char last_sr_reply[1024];
    /*! Last event received. Only populated when **save_event is NULL */
    esl_event_t *last_event;
    /*! Last event received when called by esl_send_recv */
    esl_event_t *last_sr_event;
    /*! This will hold already processed events queued by esl_recv_event */
    esl_event_t *race_event;
    /*! Events that have content-type == text/plain and a body */
    esl_event_t *last_ievent;
    /*! For outbound socket. Will hold reply information when connect\n\n is sent */
    esl_event_t *info_event;
    /*! Socket is connected or not */
    int connected;
    struct sockaddr_in addr;
    /*! Internal mutex */
    esl_mutex_t *mutex;
    int async_execute;
    int event_lock;
    int destroyed;
} esl_handle_t;
```

它有如下函数：

- new($host, $port, $password)。该函数初始化一个新的连接，仅用于 inbound 模式。在 C 语言中定义如下：

```
ESL_DECLARE(esl_status_t) esl_connect_timeout(esl_handle_t *handle,
    const char *host, esl_port_t port,
    const char *user, const char *password, uint32_t timeout);

#define esl_connect(_handle, _host, _port, _user, _password)
    esl_connect_timeout(_handle, _host, _port, _user, _password, 0)
```

可以看出，esl_connect 只是指向 esl_connect_timeout 的一个宏，默认超时时间 0 表示永不超时。

- new($fd)。该命令用于根据一个已存在的 Socket 句柄建立一个 ESLconnection 对象。它仅用于 outbound 模式。
- socketDescriptor()。该函数返回连接的 UNIX 文件句柄。
- connected()。判断是否已连接，连接返回 1，否则返回 0。
- getInfo()。当 FreeSWITCH 使用 outbound 模式连接时，它将首先发出一个 CHANNEL_DATA 事件，getInfo 会返回该事件。在 inbound 模式中它返回 NULL。
- send($command)。向 FreeSWITCH 发送一个 ESL 命令，它不会等待接收事件，而需要明确地使用 recvEvent 或 recvEventTimed 以接收返回的事件。返回事件的类型为 api/response 或 command/reply。使用 sendRecv() 可以自动获取返回结果。对应 C 语言的定义是：

```
ESL_DECLARE(esl_status_t) esl_send(esl_handle_t *handle, const char *cmd);
```

- sendRecv($command)。在 ESL 内部，sendRecv(command) 首先调用 send(command)，然后调用 recvEvent() 并最终返回一个 ESLevent 对象。recvEvent() 会在一个循环中调用并一直阻塞直到收到头域为 api/response 或 command/reply 的事件为止。在此期间所有收到的其他的将保存到一个内部队列中，这些队列中的事件会能在后续的 recvEvent() 中取到。C 语言中的定义如下：

```
ESL_DECLARE(esl_status_t) esl_send_recv_timed(esl_handle_t *handle, const char
            *cmd, uint32_t ms);
#define esl_send_recv(_handle, _cmd) esl_send_recv_timed(_handle, _cmd, 0)
```

- api($command[, $arguments])。向 FreeSWITCH 发 API 命令，它是阻塞执行的。它与 sendRecv("api $command $args") 是等价的。
- bgapi($command[, $arguments])。后台执行 API，要执行的 API 将在新的线程中执行，因而不会阻塞。该函数与 sendRecv("bgapi $command $args") 也是完全等价的。它执行后也返回一个 Job-UUID，与我们上面讨论的 bgapi 使用场景类似。
- sendEvent($event)。向 FreeSWITCH 发送一个事件。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_sendevent(esl_handle_t *handle, esl_event_t *event);
```

- recvEvent()。从 FreeSWITCH 中接收事件，如果此时没有事件，它将一直阻塞直到有新事件到来。如果在调用它之前曾经调用了 sendRecv()，并且 sendRecv() 曾经将收到的事件放到队列中，则该函数会返回队列中的第一个事件，否则，它会一直等待。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_recv_event(esl_handle_t *handle, int check_q,
esl_event_t **save_event);
```

- recvEventTimed($milliseconds)。该函数与 recvEvent 类似，不同的是它不会永远阻塞，而是将在参数指定的毫秒数会返回。recvEventTimed(0) 可以立即返回，可以用于事件轮循。

```
ESL_DECLARE(esl_status_t) esl_recv_event_timed(esl_handle_t *handle,
    uint32_t ms, int check_q, esl_event_t **save_event);
```

- filter($header, $value)。类似上面提到的 filter 命令，用于过滤事件。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_filter(esl_handle_t *handle, const char *header,
const char *value);
```

- events($event_type, $value)。订阅事件，类似上面的 event 命令。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_events(esl_handle_t *handle, esl_event_type_t
etype, const char *value);
```

- execute($app[, $arg][, $uuid])。执行 Dialplan App，并阻塞等待返回。它将最终返回一个 ESLevent 对象，通过 getHeader("Reply-Text") 方法可以获取返回值，一般来说"+OK [成功的信息]"表示执行成功，"-ERR [错误信息]"表示执行失败。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_execute(esl_handle_t *handle,
    const char *app, const char *arg, const char *uuid);
```

- executeAsync($app[, $arg][, $uuid])。与 execute() 相同，但非阻塞。两者实际上都调用了上面的 execute 命令，只是 executeAsync 带了"async: true"头。
- setAsyncExecute($value)。强制将 Socket 连接设为异步模式。value 为 1 为异步，0 为同步。调用该函数后，所有使用 execute() 执行的 App 都将带有"async: true"头域，因而执行是异步的。除此之外本函数对其他函数没有影响。在 C 语言中可以通过设置 esl_handle 结构体成员 async_execute = 1 来实现。
- disconnect()。主动中断与 FreeSWITCH 的 Socket 连接。C 语言定义如下：

```
ESL_DECLARE(esl_status_t) esl_disconnect(esl_handle_t *handle);
```

- setEventLock($value)。使用该选项后，所有后续的 execute() 调用都将带有"event-lock: true"头域。

关于 ESL 我们就先讲到这里，更多的请参考：http://wiki.freeswitch.org/wiki/Event_Socket_Library。

18.4 事件

在本章结束之前，我们再来讨论一些常用的事件。在第 5 章，我们曾简单讲过事件的一些基本知识。其实有了前面的基本知识，再加上本章的内容，笔者觉得已经讲得足够多了。讲得再多也代替不了读者自己的练习和实践，毕竟，笔者在学习这些知识的时候就是一点一点积累起来的。然而，笔者还是经常被问到这样的问题：FreeSWITCH 都会产生哪些事件啊？这些事件都是在什么情况下产生的？FreeSWITCH 的事件那么“长”，里面的都是什么意思啊？

所谓“授人以鱼不如授人以渔”，在此笔者不想直接回答这几个问题，而是先就这几个问题谈一下自己的看法。

18.4.1 事件的学习方法

首先，FreeSWITCH 事件的机制就是在特定的情况下产生特定的事件。这些事件都是在源代码的 switch_types.h 文件中定义的⊖。从定义一般能很直观地看到事件的含义。但要直观地看到事件含义需要两个基础：

- **英语要过关**。事件的定义都是很直观的英文单词，只要认识，一般就能知道该事件是什么意思。
- **FreeSWITCH 使用要熟练**。只有熟练掌握了前面章节中讲到的各种知识，多多进行实际的配置练习和拨打测试，才能比较好的理解这些事件。否则，即使源代码中所有的事件定义或注释是中文的，你也看不懂。

其次，只要了解了这些事件的含义，那么可能在什么情况下产生什么样的事件就能大体猜出，剩下的只需要去实践中验证。比如，跟踪所有的事件，然后打个电话看一下产生的事件是否跟你想的一样。

再次，要大胆。对陌生的东西不要存畏惧心理。FreeSWITCH 中的某些事件确实很长，Channel 相关的事件动辄几百行。但是，正如毛主席所说——一切反动派全是纸老虎。其实，Channel 相关的事件中比较重要的字段跟 info App 的输出都是相同的，且它们的含义我们在表 6-2 中就已写清楚了。理解了这些字段的含义，举一反三，就能很容易理解其他的了。

最后，踏实。多动手实验。爱因斯坦说过：天才是百分之一的灵感加百分之九十九的汗水。一定不要好高骛远。好多人刚学习 FreeSWITCH 几天就想编个呼叫中心的系统出来；还没学会用 FreeSWITCH，就看 FreeSWITCH 的源代码想进行二次开发。笔者认为这些做法都是不可取的。

总之，笔者推荐的学习方法是——利用自己学到的知识，跟踪所有的事件，然后打一个电话看看在不同的阶段都会产生哪些不同的事件，试着去理解事件内部内容的含义。积少成

⊖ 考虑到使用 ESL 的开发者并不一定需要去读源代码，因此，本书在附录中列出了所有的事件。并对一些关键的事件和常用事件添加了中文注释。

多，慢慢就理解了。另外，为了帮助读者跟踪调试这些事件，笔者也在 FreeSWITCH 中写了一些图形化的跟踪调试的工具，它们的使用方法将在下一章最讲到。

学而不思则罔，思而不学则殆。到这里，读者不妨合上书，思考一下我们刚刚谈到的这些有没有道理？你是否其中具备上面提到的两个基础？你都是如何学习的？其实，即使英语水平不好也无妨，多查查字典，所有事件加起来也没有多少单词；但是，如果 FreeSWITCH 使用不熟练的话，就说明你没有好好下工夫了。学习的态度永远比能力重要。

18.4.2 常用事件简介

上面啰啰嗦嗦讲了那么多，我们还是言归正传。简单介绍一下事件的种类和产生的场景。

我们曾在第 5 章讲到 FreeSWITCH 中的事件分为主事件和 CUSTOM 事件，但其实 FreeSWITCH 中也没有严格的分类方法。下面我们就以 CUSTOM 事件、CHANNEL 事件、CHANNEL 相关的事件、系统事件和其他事件这种分类来进行说明。

1. CUSTOM 事件

CUSTOM 事件主要是用于跟系统内部核心逻辑关系不大的一些事件，一般用于模块内部，且便于扩展。它的 Event-Name 永远是 CUSTOM，不同事件的类型使用 Event-Subclass 区分。Event-Subclass 可以是任意的字符串，可以任意定义，但 FreeSWITCH 中也有约定俗成的命名空间，如 mod_sofia 模块中常用的 sofia::register、sofia::unregister 等，都使用以“::”隔开、以 sofia 开头的命名空间。类似的还有，会议模块（mod_conference）中的 conference::maintenaince、FIFO 模块（mod_fifo）中的 fifo:info 等。

2. Channel 事件

有一部分事件是以 Channel 开头的，它主要跟 Channel 的状态（状态机）有关。下从以 Channel 的生命周期来大体讲一下。

首先，系统中有来话或去话时，将生成一个新的 Channel，并产生一个 CHANNEL_CREATE 事件。该事件中会包含该 Channel 的 UUID（其对应的字段名字是 Unique-ID）、主叫号码、被叫号码、Channel 的创建时间等。

接下来，如果 Channel 正常继续进行，则会产生 CHANNEL_PROGRESS 事件（如在 SIP 中收到对方的 100 或 180 消息）。如果在应答之前能收到 Early Media（如在 SIP 中收到对方的）183 消息，则会产生 CHANNEL_PROGRESS_MEDIA 事件。

如果一个 Channel 被应答，就会产生 Channel Answer 事件。

如果一个 Channel 与另外一个 Channel bridge 成功，则会产生 CHANNEL_BRIGE 事件。注意，bridge 是由两个 Channel 参与的，其中两个 Channel 分别称为 a-leg 和 b-leg。在 SIP 中，如果使用 bridge 这个 App 等待 bridge 的 Channel 收到对端 Channel 发来的 Early Media 消息（如 SIP 中的 183 消息）即 bridge 成功，可以在 Channel 创建时使用 ignore_early_media 通道变量延迟 bridge 的返回（直到应答，如收到 SIP 中的 200 消息。但结果是 a-leg，则听不到 b-leg 发来的 Early Media（回铃音）。

另外，CHANNEL_BRIDGE 事件只在一个 Channel 上发生，即只发生在主动的那个 leg 上。例如，终端 A 通过 FreeSWITCH 呼叫 B，A 端的呼叫到达 FreeSWITCH 后产生一个新的 Channel，FreeSWITCH 使用 bridge App 去呼叫 B，又产生一个新的 Channel。如果 B 应答，则 b-leg 上会产生一个 CHANNEL_ANSWER 消息，同时，该应答信号通过 bridge App 传递到 A 上，a-leg 上也会产生一个 CHANNEL_ANSWER 消息。bridge 成功后（假设没有 Early Media 参与，因此会直到应答 bridge 才完成），a-leg 上会产生 CHANNEL_BRIGE 事件，而 b-leg 上不会。

CHANNEL_BRIGE 事件与上面单腿的事件不同的地方在于它里面包含了 b-leg 的信息，如 Other-Leg-Unique-ID 是 b-leg 的 Channel UUID，Other-Channel-Caller-ID-Number 是 b-leg 上的主叫号码等。

挂机后将产生 CHANNEL_HANGUP 事件和 CHANNEL_HANGUP_COMPLETE 事件，其中，后者比前者内容丰富一些，比如后者带有 variable_duration（通话时长，从 Channel 创建开始计时）及 variable_billsec（计费时长，从应答后开始计时）。因此，一般使用该事件取计费信息。

最后还将产生 CHANNEL_DESTROY 事件，表示该 Channel 已经完全释放（销毁）了。

在 Channel 生存期间，在执行 App 的时候，将产生 CHANNEL_EXECUTE 事件，表示一个 App 已开始执行。有的 App 执行非常快，如 set；有的则可能比较慢，如 playback（要等声音文件播放完毕）。在 App 执行完毕后，会产生 CHANNEL_EXECUTE_COMPLETE 事件。在使用异步方式进行 ESL 编程时可以在收到该事件后执行下一个 App。其中，这两个事件都会包含一个 Application 字段，标志当前正在执行或已完成的 App 的名字。

对于大部分的 Channel 事件，与 Channel 相关的 Channel Variable 都会附加在 Channel 相关的事件上。与系统内部的字段名字不同（大写字母开头加中横线方式命名，如 Unique-ID），这些 Channel Variable 都是以“variable_”开头的，如 variable_effective_caller_id_name。

某些 Channel 事件默认没有 variable_ 开头的字段，如 CHANNEL_CALLSTATE。这主要是为了减少消息量。如果希望这些消息中也包含所有的 variable，则可以在 Dialplan 中使用 verbose_events App 来打开，如：

```
<action application="verbose_events" data="true"/>
```

3. Channel 相关事件

有一部分事件虽然与 Channel 相关的，但是它们的名字不是以 CHANNEL_ 开头的，例如 PLAYBACK_START（放音开始）、PLAYBACK_STOP（放音结束）、RECORD_START（录音开始）、RECORD_STOP（录音结束）、DTMF（双音多频按键信息）等。

这类事件也有 Channel UUID（Unique-ID），但它们一般与 Channel 的状态无关。

4. 系统事件

系统事件包含 STARTUP（系统启动）、SHUTDOWN（系统关闭）、MODULE_LOAD（模

块加载）、MODULE_UNLOAD（模块卸载）等。另外，系统每隔 20 秒会产生一个 HEARTBEAT（心跳）事件，可以用于检测 FreeSWITCH 是否正常运行。

其他还有许多事件，如 API（执行 API 时产生）、BACKGROUND_JOB（使用 bgapi 后台执行 API 时产生）等，在此我们就不多介绍了。

18.5 小结

本章详细介绍了 Event Socket 协议、ESL 库及库函数的使用方法等，并通过一些简单的例子讲了它们在编程开发中的使用方法和注意事项。

初学者往往比较容易理解 Event Socket 的 Inbound 模式和 Outbound 模式，但对于何时该使用哪种模式不是很清楚。一般来说，Outbound 模式与上一章讲的嵌入式语言相比更强大，它比较适合控制单腿的呼叫，实现复杂的 IVR 应用；而 Inbound 更适合接收所有的事件，与多条腿进行交互，进行更复杂的呼叫控制。其中，在 Outbound 模式中，又分为同步模式和异步模式，同步模式控制比较简单，但自由度较小；异步模式需要更多的编程技巧，但会更强大。

当然，上述说法不是绝对的，Inbound 和 Outbound 两种模式都能完成所有的控制功能，只是，Outbound 模式的 Socket 是由 FreeSWITCH 建立的，它是建立在 Channel 的基础上的，每一个 Channel 均会为外部的 TCP Server 建立一个连接，在 Channel 挂机时释放。因此，Outbound 的连接要考虑 Channel 的生命周期（即 Socket 的生命周期）。而 Inbound 的连接由客户端主动向 FreeSWITCH 发起连接，极需要考虑断线重连等问题。

在实际开发应用中，具体使用哪种模式需要具体问题具体分析，解决问题需要自己动手进行实验，而不能盲目迷信别人说的。

如果你使用 C 语言进行 ESL 开发，可以参考一下源代码目录中的 fs_cli.c（在 libs/esl 目录中），里面有各种函数的真实使用方法。我们也将在下一节再讲解一些实际的例子。

在本章的最后，我们也讲了 FreeSWITCH 的事件并重点讲了学习方法。后面讲了一些常用的事件主要是给想“偷懒”的读者一个捷径，以快速理解和使用这些事件，但更深入的学习还需要读者自己实际练习。

第 19 章

使用 ESL 开发

上一章我们学习了 ESL，本章我们就趁热打铁再继续讲一些与 ESL 相关的例子。这些例子有的是从真实的项目中精简出来的，有的是为了讲了某些概念和方法精心编写的。我们在讲解的过程中也尽量以实际项目开发的方式将源代码纳入版本管理系统，并逐步修改和完善，以便读者更容易理解。ESL 是一个客户端库，它主要用于对 FreeSWITCH 进行逻辑控制，因此，很多实际的功能还得靠我们前面讲过的 App 和 API 来完成。因此，如果读者在学习过程中注意复习和强化我们前面学习过的基本概念，并比较它与在 Dialplan 中及嵌入式脚本中使用方法的异同，学习本章必能事半功倍。同时，本章中的开发方式和方法也有助于理解和学习后面的章节，希望读者认真阅读和实践。

19.1 创建独立的 ESL 应用

在上一章，我们也讲了一些 ESL 的例子，但为了简单起见都是直接在 FreeSWITCH 源代码提供的例子上改的，也没有写自己的 Makefile，这样有的读者在自己做项目的时候可能会觉得无从下手。下面我们就把 ESL 这一块从 FreeSWITCH 中独立出来。

当然，为了跟给读者讲清楚其中的原理，我们独立还是一步一步地进行。在进行下一步之前，我们先要为我们的新的项目起个名字，就叫 myesl 吧，虽然这个名字没有什么创造性。

19.1.1 创建目录和源文件

下面我们先尝试把实现我们自己业务逻辑的源文件分离出来。首先，创建一个叫 myesl 的目录，把我们相关的源代码都放到该目录里面。

然后，创建 myesl.c 文件，它的内容就直接把 FreeSWITCH 源码目录下的 testclient.c 中的内容原样复制过来即可，我们暂时不改源代码，主要是看一看怎么把它独立出来。既然我们原文件已经有了，下面就写一个 Makefile，保存到相同的目录下，内容如下：

```
ESLPATH = /usr/src/freeswitch/libs/esl
```

```
CFLAGS   = -I$(ESLPATH)/src/include

all: myesl.c $(ESLPATH)/libesl.a
    gcc $(CFLAGS) -o myeslmyesl.c $(ESLPATH)/libesl.a
```

其中，第1行在Makefile中定义一个变量，它即是ESL库所在的路径（笔者把FreeSWITCH的源代码放到/usr/src/freeswitch目录中）。第2行定义另一个变量，使用“-I”参数指定C语言头文件的位置，这一行还使用“$()”引用了第1行中定义的变量ESLPATH。

第3行中的all，是一个目标（Target）。在Makefile中，默认会编译第一个目标，该目录依赖于myesl.c以及ESL的静态库文件libesl.a（所以libesl.a是必须存在的，即FreeSWITCH必须事先被编译过）。这样写的好处是，如果myesl.c已经编译过，那么在下次编译的时候就会检查它有没有变化，如果没有变化，则不用重新编译。这样做在单独一个源文件的时候影响不是很大（重新编译一次也费不了多少时间），但在编译比较大型的项目时，可能会有很多源文件，那么节省的时间就很可观了。

真正的编译动作是在第4行执行的，它调用gcc进行编译。命令行参数中，使用了“$(CFLAGS)”指定了头文件的位置；“-o”后面指定编译完成后生成的可执行文件的名称；myesl.c即是源文件；最后一个参数指定了ESL静态库的位置（libesl.a，注意ESL默认只生成静态库）。

19.1.2 编译和执行

一切准备好以后，接下来直接在控制台上输入make命令进行编译，编译完成后将产生myesl可执行程序，然后即可以直接执行看一下效果。下面是执行make以及我们新生成的myesl命令的输出结果：

```
$ make
gcc -I/usr/src/freeswitch/libs/esl/src/include -o myeslmyesl.c /usr/src/
freeswitch/libs/esl/libesl.a

$ ./myesl
UP 0 years, 0 days, 4 hours, 26 minutes, 48 seconds, 650 milliseconds, 344 microseconds
FreeSWITCH (Version 1.5.8b git e7fe9aa 2013-11-25 00:45:51Z 64bit) is ready
0 session(s) since startup
0 session(s) - peak 0, last 5min 0
0 session(s) per Sec out of max 200, peak 0, last 5min 0
3000 session(s) max
min idle cpu 0.00/100.00
Current Stack Size/Max 240K/8192K
```

至此，便大功告成了。我们成功地把我们的ESL应用与FreeSWITCH源代码分隔开了，以后再也不用跟FreeSWITCH的源代码放到同一目录了。其实，编译ESL应用程序时，不需要依赖全部FreeSWITCH的源代码了，只需要依赖于ESL的头文件及静态库文件libesl.a。也就是说，ESL的源代码是可以单独存在的，可以将FreeSWITCH源代码目录中的libs/esl目录复制到别的地方，以后做ESL客户端应用时就不需要全部的FreeSWITCH的源代码了。

19.1.3 将源代码纳入版本控制

在开发过程中，我们经常会对源代码进行修改，因此需要有版本控制工具进行管理。FreeSWITCH 是使用 Git 进行源代码管理的，大家在前面的学习中也熟悉了 Git 的复制操作，下面我们再带领大家熟悉一下其他的操作，到后面我们阅读 FreeSWITCH 源代码甚至进行 FreeSWITCH 开发的时候，将会用到更多的 Git 功能。

使用 Git 很简单，只需要在我们的 myesl 目录中执行如下三条命令：

第一条，初始化一个版本库，该版本库仅在当前目录中有效。

第二条，把我们新建的两个文件增加到版本库管理中来。

第三条，提交（commit）我们的修改，这两个文件便存入版本库，并产生一条提交历史。

具体的命令如下：

```
$ git init
$ git add Makefilemyesl.c
$ git commit -m 'initial commit' .
```

与 SVN 等版集中式的代码管理工具不同，Git 的管理是分布式的，因此上述的提交操作仅在本地执行，如果需要与其他人共享，便需要将本地的代码库“推”（push）到远端去。Github 是一个很好的源代码管理平台，对于开源项目是免费的。为了更好地与各位读者分享，我们也将我们的代码“推”上去。

具体的 Github 操作在此就不多讲了，笔者在 Github 上创建了一个远程版本库，它的地址是 https://github.com/seven1240/myesl.git，下面首先在我们本地的代码库中用如下命令增加远程版本库：

```
$ git remote add origin https://github.com/seven1240/myesl.git
```

然后，就可以“推”了，命令和显示结果如下：

```
$ git push origin master
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 658 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/seven1240/myesl.git
 * [new branch]      master -> master
```

推送成功后，就可以用浏览器访问 https://github.com/seven1240/myesl 来查看远程版本库的情况了[⊖]。

读者感兴趣的话，也可以把该项目复制到本地计算机上，如下面是笔者测试的复制过程：

```
dujinfang@seven:~$ cd /tmp
dujinfang@seven:/tmp$ git clone https://github.com/seven1240/myesl.git
Cloning into 'myesl'...
remote: Counting objects: 7, done.
```

⊖ 在首次查看的时候，该页面建议增加一个 README，以让读者知道该项目是干什么的，此时读者就可以单击“创建”按钮在上面的网页上自动创建了。

```
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (7/7), done.
```

19.2 用 ESL 重写空中充值服务

在 17.3 节我们讲过一个空中充值服务，当时是用 Lua 脚本实现的，下面我们来看一下在 ESL 中该怎么实现。

IVR 类应用一般只涉及一个 Channel，因此比较适合使用 ESL 的 Outbound 模式——即我们启动一个 ESL 服务器，当有新的呼叫到来时 FreeSWITCH 便会连接我们的 ESL 服务器。

在此，我们还是使用 C 语言作为例子。接着上一节的内容，在 myesl 目录中创建源文件 charge.c。我们稍后再看如何改写 Makefile，下面我们会先从代码看起。

注意

一般来说，在 C 语言程序中，被调用的函数写在前面，调用别的函数的函数则写在后面，这样就可以避免对所有的被调用的函数做前向声明。而在我们的程序中，main 要调用前面的其他函数，所以我们把 main 函数写到最后。但从逻辑上看，程序是从 main 函数开始执行的，所以我们先从 main 函数开始看。

下面是我们的 main 函数入口，它比较简单。第 188 行定义 main 函数。第 190 行设置 ESL 日志级别，在此我们使用的是 INFO 级别，如果在调试的情况下，可以将日志级别改为 ESL_LOG_LEVEL_DEBUG，这样便可以在控制台上看到大量日志输出。第 191 行使用 esl_listen_threaded 函数监听本地的一个 8040 端口，并阻塞等待连接到来。如果有新的连接到来，它就会启动一个新线程，并回调 charge_callback 函数⊖。

```
188  int main(void)
189  {
190    esl_global_set_default_logger(ESL_LOG_LEVEL_INFO);
191    esl_listen_threaded("localhost", 8040, charge_callback, NULL, 100000);
192    return 0;
193  }
```

charge_callback 回调函数是在第 161 行定义的，当有新的连接到来时，它便在新的线程里执行。在函数的一开始，我们初始化了一个 handle 变量，用于标识一个 ESL Socket 连接，以及一个 charge_helper 结构体变量，用于记录充值过程中的一些数据。

```
161   static void charge_callback(esl_socket_t server_sock, esl_socket_t
client_sock, struct sockaddr_in *addr, void *user_data)
162  {
163    esl_handle_t handle = {{ 0 }};
164    esl_status_t status;
```

⊖ 注意，这段代码是基于 FreeSWITCH 1.4 版（Git master）的例子。回调函数中多了一个私有用户数据（user_data）参数，允许在 main 函数和回调函数间传递私有数据，虽然本例中我们没有用到私有数据。参见 18.3.1 节中 C 语言的例子。

```
165    charge_helper_t charge_helper = { 0 };
```

接下来初始化 charge_helper 的值。第 166 行设置 handle。第 167 行初始化当前的余额。第 168 行将充值状态机的当前状态设为 CHARGE_WELCOME，即欢迎主菜单。

```
166    charge_helper.handle = &handle;
167    charge_helper.balance = BALANCE;
168    charge_helper.state = CHARGE_WELCOME;
```

第 169 行将连接进来的 Socket（client_sock）与我们这里的 handle 进行绑定，以后所有与 FreeSWITCH 的交互都通过该 handle 进行。第 171 行订阅我们需要的 CHANNEL_EXECUTE_COMPLETE 事件。第 172 行通过一个过滤器（filter）告诉 FreeSWITCH，我们仅需要关注与本次通话（当前的 Channel）相关的事件，而其他的事件一概不关心。注意该过滤器的作用不是过滤掉，而是“过滤进”，即过滤关键字（这里是 Unique-ID）指定的是想要的内容，而不是不想要的内容。当然，为了取得当前通话的 UUID，我们使用了 esl_event_get_header 函数从 handle.info_event 中取出。在有通话到来时，FreeSWITCH 会首先发一个事件告诉我们的 ESL 程序该通话相关的信息，这些信息就存在 handle.info_event 变量中。另外，第 173 行我们还给 FreeSWITCH 发送 linger API 命令，通知 FreeSWITCH 如果电话挂机的话，“逗留”5 秒后再将该 Socket 断开，以保证所有我们需要的事件都能收到。

```
169    esl_attach_handle(&handle, client_sock, addr);
170    esl_log(ESL_LOG_INFO, "Connected! %d\n", handle.sock);
171    esl_events(&handle, ESL_EVENT_TYPE_PLAIN, "CHANNEL_EXECUTE_COMPLETE");
172    esl_filter(&handle, "Unique-ID",
           esl_event_get_header(handle.info_event, "Caller-Unique-ID"));
173    esl_send_recv(&handle, "linger 5");
174    esl_log(ESL_LOG_INFO, "%s\n", handle.last_sr_reply);
```

本函数中上面的语句都是当有电话进来时进行的一些初始化和准备工作，从第 175 行开始，就通过 esl_execute 函数在 FreeSWITCH 中执行一些 App 了，这些 App 的执行跟在 Dialplan 中是类似的。如第 175 行的 answer 表示对来话进行应答（必须先应答再放音），然后设置中 TTS 引擎的参数（第 176 ~ 177 行），并播放欢迎音（第 178 行）。

```
175    esl_execute(&handle, "answer", NULL, NULL);
176    esl_execute(&handle, "set", "tts_engine=tts_commandline", NULL);
177    esl_execute(&handle, "set", "tts_voice=Ting-Ting", NULL);
178    esl_execute(&handle, "speak", "您好，欢迎使用空中充值服务", NULL);
```

下面是一个 while 循环，不断地从 FreeSWITCH 接收（esl_recv）事件，收到的事件将存到变量 handle.last_event 中，如果我们判断事件的类型（type）是“text/event-plain”（第 181 行），说明是一个我们订阅的事件，然后就可调用回调函数 event_callback（第 182 行）进行处理。注意，这里我们收到的事件 last_event 是 ESL 对 FreeSWITCH 中事件的一个包装，我们需要的真正的事件在 last_ievent 中，所以后面将看到在第 56 行我们将事件指针 event 指向 handle->last_ievent。

```
179    while((status = esl_recv(&handle)) == ESL_SUCCESS) {
180      const char *type = esl_event_get_header(handle.last_event, "content-type");
```

```
181       if (type && !strcasecmp(type, "text/event-plain")) {
182         event_callback(&charge_helper);
183       }
184     }
```

如果上述 while 循环中接收失败（可能是 Socket 已断开），则循环结束，我们打印一行日志（第 185 行），并为了保险起见再调用一次 esl_disconnect（第 186 行）断开 Socket 连接。该线程会退出。

```
185     esl_log(ESL_LOG_INFO, "Disconnected! %d\n", handle.sock);
186     esl_disconnect(&handle);
187   }
```

可以看出，系统的关键部分在 event_callback 函数中。该函数我们稍后会讲到，现在为了理解该函数，我们从返回到文件开始处，从程序的第一行开始看。

在程序的一开始，先装入必要的头文件（第 1 ~ 3 行）以及定义几个宏（第 4 ~ 13 行）。其中，ENSURE_INPUT 宏（第 8 ~ 13 行）比较长，它用于检查用户的输入是否为空，以避免遇到空指针引起崩溃。

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <esl.h>

04  #define ERROR_PROMPT "say:输入错误，请重新输入"
05  #define BALANCE 100
06  #define CHARGE  100
07  #define set_string(dest, str) strncpy(dest, str, sizeof(dest) - 1)
08  #define ENSURE_INPUT(input) if (!input) { \
09    esl_execute(ch->handle, "speak", "再见", NULL); \
10    /* sleep(1); */ \
11    esl_execute(ch->handle, "hangup", NULL, NULL); \
12    return; \
13  }
```

接下来，我们定义几个结构。charge_menu_t 是一个 enum 的类型，用于定义菜单项。其中，为了与用户输入按键相匹配，我们定义了一个无效的菜单 CHARGE_MENU_NONE，它的值为 0，那么后面的 CHARGE_MENU_QUERY 和 CHARGE_MENU_CHARGE 的值就可以与菜单项相应的按键（1 和 2）对应起来。使用 enum 类型的常量在 C 语言编程中是一个好的习惯，这样代码就比较易读。后面接着我们定义了一个 charge_state_t 类型，作为菜单中的状态机使用（注意由于 C 语言的编程方法不太一样，这里跟 Lua 脚本中的状态并不是一一对应的），它也是对 enum 的重定义。

```
14  typedef enum charge_menu_s {
15    CHARGE_MENU_NONE,
16    CHARGE_MENU_QUERY,
17    CHARGE_MENU_CHARGE
18  } charge_menu_t;

19  typedef enum charge_state {
20    CHARGE_WELCOME,
```

```
21     CHARGE_MENU,
22     CHARGE_WAIT_ACCOUNT,
23     CHARGE_WAIT_ACCOUNT_PASSWORD,
24     CHARGE_WAIT_CARD,
25     CHARGE_WAIT_CONFIRM
26   } charge_state_t;
```

下面定义一个 charge_helper_t 类型，用于保存充值环节中数据。

```
27   typedef struct charge_helper_s {
28     esl_handle_t *handle;
29     charge_state_t state;
30     charge_menu_t menu;
31     char account[20];
32     char card[20];
33     int balance;
34   } charge_helper_t;
```

get_digits 函数用于从用户输入事件中获取输入的 DTMF 字符串。

```
35   char * get_digits(esl_event_t *event)
36   {
37     char *digits = esl_event_get_header(event, "variable_digits");
38     if (digits) esl_log(ESL_LOG_INFO, "digits: %s\n", digits);
39     return digits;
40   }
```

下面的 check_account_password 和 check_account_card 函数只是用于硬编码的检查用户的账号、卡号、密码等⊖。do_charge 函数也只是简单地将余额与充值金额相加。这些函数在以后都可以扩充（如与数据库交互等）。

```
41   int check_account_password(char *account, char *password)
42   {
43     return (!strcmp(account, "1111")) && (!strcmp(password, "1111"));
44   }

45   int check_account_card(char *account, char *card)
46   {
47     return (!strcmp(account, "1111")) && (!strcmp(card, "2222"));
48   }

49   int do_charge(int balance, int charge)
50   {
51     return balance + charge;
52   }
```

下面的 event_callback 函数（第 53 行）是一个回调函数，我们在每次收到相关的事件时，都会回调该函数。在该函数内，我们实现了一个简单的状态机，用于根据用户的输入进行各种判断，以决定执行哪个流程。

该函数的输入参数是一个 charge_helper_t 类型的指针 ch，通过它可以取得相应的收到

⊖ 注意，strcmp 函数在匹配成功时返回 0（在 C 语言逻辑判断中为“假”），因而习惯用法是在前面加上一个“!”对 0 取反使之为“真”。这是该函数容易造成困扰的地方，好多 Bug 也是因此产生的。但这是一个不可避免的事实。

的事件，如第 56 行将 event 指针指向当前收到的事件。第 57 行是一行调试信息，我们暂时注释掉了，在用到的时候可以将注释去掉，以方便调试。第 58 行判断事件的类型，我们只接收 CHANNEL_EXECUTE_COMPLETE 类型的事件，其他事件一概不理。第 59 行我们将从当前事件中取得最后执行的 App 的名字，并赋值给 application 变量。

```
53  static void event_callback(charge_helper_t *ch)
54  {
55    char *application = NULL;
56    esl_event_t *event = ch->handle->last_ievent;

57    /* esl_log(ESL_LOG_INFO, "event_id: %d\n", event->event_id); */

58    if (event->event_id != ESL_EVENT_CHANNEL_EXECUTE_COMPLETE) return;

59    application = esl_event_get_header(event, "Application");
60    esl_log(ESL_LOG_INFO, "State: %d App: %s\n", ch->state, application);
```

第 61 行用一个 switch 语句判断当前的状态。电话刚刚接入时将进入 CHARGE_WELCOME 状态，然后我们判断上一次执行的 App 是不是 speak（注意，这里我们只有一个 speak，如果前面执行多个 speak，则需要更多的条件进行判断，以找到最后一个 speak），如果是，则说明前面的欢迎音（第 180 行，后面会讲到）播放完了，便接着使用 play_and_get_digits 提示用户选择是查询还是充值，以进入相应的菜单。注意这里的 play_and_get_digits 与 Lua 中的用法有点差别，前者将收到的输入结果存入 digits 变量中，我们后续收到 CHANNEL_EXECUTE_COMPLETE 时便可以从事件的 variable_digits 头域中取得结果（如第 37 行所示）。当然，在这里我们只需要使用 esl_execute（第 66 行）通知 FreeSWITCH 执行就行了，执行完毕后我们将在下次收到事件时获得用户输入。当然，我们事先在第 65 行将下一次的状态设为 CHARGE_MENU，这样在下次收到事件后，我们将跳到第 71 行开始判断。

```
61    switch(ch->state) {
62      case CHARGE_WELCOME:
63        if (!strcmp(application, "speak")) {
64  select_menu:
65          ch->state = CHARGE_MENU;
66          esl_execute(ch->handle, "play_and_get_digits",
67            "1 1 3 5000 # 'say:查询请按1，充值请按2' "
68            ERROR_PROMPT " digits ^\\d$", NULL);
69        }
70        break;
71      case CHARGE_MENU:
72        if (!strcmp(application, "play_and_get_digits")) {
73          char *menu = get_digits(event);
```

在收到用户输入后，我们将使用 ENSURE_INPUT 宏判断用户输入是否合法，如果不合法则立即播放“再见”并挂机。

```
74          ENSURE_INPUT(menu)

75          if (!strcmp(menu, "1")) {
```

```
76            ch->menu = CHARGE_MENU_QUERY;
77          } else if (!strcmp(menu, "2")) {
78            ch->menu = CHARGE_MENU_CHARGE;
79          }
```

在获取到用户输入的菜单后（1或2），我们接着询问用户的账号（第81行），并将状态设为CHARGE_WAIT_ACCOUNT（第80行）等待下一次事件到来。

```
80          ch->state = CHARGE_WAIT_ACCOUNT;
81          esl_execute(ch->handle, "play_and_get_digits",
82            "4 5 3 5000 # 'say:请输入您的账号，以井号结束' "
83            ERROR_PROMPT " digits ^\\d{4}$", NULL);
84        }
85        break;
86      case CHARGE_WAIT_ACCOUNT:
87        if (!strcmp(application, "play_and_get_digits")) {
88          char *account = get_digits(event);

89          ENSURE_INPUT(account)

90          set_string(ch->account, account);
```

当用户输入账号后，检查用户在上一步中选择的菜单项（已存到ch->menu变量中），并根据不同的菜单项决定下一步继续询问密码还是询问卡号。

```
91          if (ch->menu == CHARGE_MENU_QUERY) {
92            ch->state = CHARGE_WAIT_ACCOUNT_PASSWORD;
93            esl_execute(ch->handle, "play_and_get_digits",
94              "4 5 3 5000 # 'say:请输入您的密码，以井号结束' "
95              ERROR_PROMPT " digits ^\\d{4}$", NULL);
96          } else if (ch->menu == CHARGE_MENU_CHARGE) {
97            ch->state = CHARGE_WAIT_CARD;
98            esl_execute(ch->handle, "play_and_get_digits",
99              "4 5 3 5000 # 'say:请输入您的充值卡卡号，以井号结束' "
100              ERROR_PROMPT " digits ^\\d{4}$", NULL);
101         } else {
102           ch->state = CHARGE_WELCOME;
103           esl_execute(ch->handle, "speak", "输入有误，请重新输入", NULL);
104         }
105       }
106       break;
```

在等待账户密码状态下，如果收到用户输入事件，则检查用户输入的账户、密码是否匹配（第111行）。如果匹配，则将下一步的状态设为CHARGE_WELCOME（第114行），以使播放完毕后重新进入主菜单，并播放当前余额（第115行）；如果用户输入错误（第116行），则播放错误提示，也转入主菜单。

```
107     case CHARGE_WAIT_ACCOUNT_PASSWORD:
108       if (!strcmp(application, "play_and_get_digits")) {
109         char *password = get_digits(event);

110         ENSURE_INPUT(password)

111         if (check_account_password(ch->account, password)) {
```

```
112             char buffer[1024];
113             sprintf(buffer, "您的余额是 %d 元", ch->balance);
114             ch->state = CHARGE_WELCOME;
115             esl_execute(ch->handle, "speak", buffer, NULL);
116           } else {
117             ch->state = CHARGE_WELCOME;
118             esl_execute(ch->handle, "speak", "输入有误，请重新输入", NULL);
119           }
120         }
121         break;
```

如果用户当初选择了充值，则下面这一步将得到卡号（第 124 行）。如果用户输入的卡号是正确的（第 126 行），则播放提示音提示充值卡上的金额（第 129 行）。注意其中我们在第 127 行初始化了一个缓冲区，并在第 128 行写入一个字符串，这是 C 语言中拼接字符串的一种方式，为了避免缓冲区溢出需要保证缓冲区足够大（这里我们使用了 1024 字节）。接下来让用户确认是否充值（第 132 行），并将下一步的状态设为 CHARGE_WAIT_CONFIRM 等待用户输入结果。如果用户输入的号码有误（第 135 行），则转入主菜单。

```
122       case CHARGE_WAIT_CARD:
123         if (!strcmp(application, "play_and_get_digits")) {
124           char *card = get_digits(event);

125           ENSURE_INPUT(card)

126           if (check_account_card(ch->account, card)) {
127             char buffer[1024];
128             sprintf(buffer, "您要充值 %d 元", CHARGE);
129             esl_execute(ch->handle, "speak", buffer, NULL);
130             /* sleep(2); */
131             ch->state = CHARGE_WAIT_CONFIRM;
132             esl_execute(ch->handle, "play_and_get_digits",
133               "1 1 3 5000 # 'say:确认请按 1，返回请按 2' "
134               ERROR_PROMPT " digits ^\\d$", NULL);
135           } else {
136             ch->state = CHARGE_WELCOME;
137             esl_execute(ch->handle, "speak", "输入有误，请重新输入", NULL);
138           }
139         }
140         break;
```

在等待用户输入状态下，如果用户按“1”确认，则进行充值（第 147）操作，并向用户播报充值结果（第 150 行），播报完毕后，就可以返回主菜单了（第 149 行）。如果用户按“2”则放弃充值并返回主菜单。这时我们就会遇到一个问题：首先看第 63 行有一个验证，即最后执行的 App 必须是一个 speak 才能顺利往下走，我们前面返回主菜单前都使用了 speak（如第 117 ~ 118 行、第 136 ~ 137 行），而此处用户输入“1”以后，显然在此播放任何语音都是不合适的，因此这里我们使用了 C 语言中的一个技巧，第 153 行使用 goto 语句直接跳转到在第 64 行定义的 select_menu 标签，并直接播放主菜单，而不用等待下一次事件的到来。虽然 goto 语句在 C 语言中一般是不推荐使用的，但是我们这里编写的是“不一般”的程序，而且到后面我们看 FreeSWITCH 源代码的时候，也能看到大量的 goto 语句。

```
141       case CHARGE_WAIT_CONFIRM:
142         if (!strcmp(application, "play_and_get_digits")) {
143           char *confirm = get_digits(event);

144           ENSURE_INPUT(confirm)

145           if (!strcmp(confirm, "1")) {
146             char buffer[1024];

147             ch->balance = do_charge(ch->balance, CHARGE);
148             sprintf(buffer, "充值成功，充值金额 %d 元，余额为 %d 元 ", CHARGE, ch->balance);
149             ch->state = CHARGE_WELCOME;
150             esl_execute(ch->handle, "speak", buffer, NULL);
151           } else if (!strcmp(confirm, "2")) {
152             ch->state = CHARGE_WELCOME;
153             goto select_menu;
154           }
155         }
156         break;
157       default:
158         break;
159     }

160   }
```

到此为止，我们充值服务的所有代码都准备好了。接下来就需要编译运行了。为了编译运行新的充值服务程序，我们需要修改我们的 Makefile，内容如下所示：

```
ESLPATH = /usr/src/freeswitch/libs/esl
CFLAGS  = -ggdb -I$(ESLPATH)/src/include

all: myesl charge

myesl: myesl.c
    gcc $(CFLAGS) -o myeslmyesl.c $(ESLPATH)/libesl.a

charge: charge.c
    gcc $(CFLAGS) -o charge charge.c $(ESLPATH)/libesl.a
```

修改后的 Makefile 有 3 个 Target——all、myesl 和 charge。其中 all 是默认的，但它又依赖于其他两个 Target。如果在当前目录中执行 Make，则它会先编译生成 myesl，再编译 charge。

编译完成后，将生成一个名为 charge 的可执行文件，它就是我们的“空中充值服务”程序。运行该程度后便监听 8040 端口，一旦有新的连接进来，便开启一个新线程为用户服务，用户挂机后线程退出。笔者拨打电话测试了查询菜单，日志输出如下（由于源文件中有空行，实际日志输出中的行号与代码中的行号不对应，为了读者参考方便对照源代码，笔者手动修改了日志中的行号）：

```
[INFO] charge.c:170 charge_callback() Connected! 4
[INFO] charge.c:174 charge_callback() +OK will linger 5 seconds
[INFO] charge.c:60 event_callback() State: 0 App: answer
[INFO] charge.c:60 event_callback() State: 0 App: set
```

```
[INFO] charge.c:60 event_callback() State: 0 App: set
[INFO] charge.c:60 event_callback() State: 0 App: speak
[INFO] charge.c:60 event_callback() State: 1 App: play_and_get_digits
[INFO] charge.c:38 get_digits() digits: 1
[INFO] charge.c:60 event_callback() State: 2 App: play_and_get_digits
[INFO] charge.c:38 get_digits() digits: 1111
[INFO] charge.c:60 event_callback() State: 3 App: play_and_get_digits
[INFO] charge.c:38 get_digits() digits: 1111
[INFO] charge.c:60 event_callback() State: 0 App: speak
[INFO] charge.c:60 event_callback() State: 1 App: play_and_get_digits
[INFO] charge.c:185 charge_callback() Disconnected! 4
```

在本例中，我们使用了ESL的Outbound模式，即我们的程序启动后便作为一个ESL服务器程序监听一个TCP端口，而FreeSWITCH中有电话到来时，便主动连接我们的服务器程序。我们的服务器程序是多线程的，因而可以同时为多个用户服务。我们使用如下的Dialplan配置将来话送到我们的充值服务器：

```
<extension name="IVR Charge">
    <condition field="destination_number" expression="^1234$">
        <action application="socket" data="127.0.0.1:8040 full"/>
    </condition>
</extension>
```

注意，在这里我们使用了ESL的“同步”模式。在同步模式中，使用esl_execute执行App时是顺序进行的，只有等待一个执行完了才会执行下一个，因而编程比较简单（缺点是不能打断正在执行的App，如想打断一个正在播放长语音文件[⊖]的App是无法实现的）。当然，有时我们可能也希望使用“异步”模式的ESL连接，那么可以在Dialplan配置中加入async参数，如：

```
<action application="socket" data="127.0.0.1:8040 async full"/>
```

当然，在使用了async参数的情况下，我们上述程序的第129行的执行就会出现问题，因为它会紧接着被132行的App抢占了。在出现这种问题时，需要等待第129行的App执行完毕后（收到CHANNEL_EXECUTE_COMPLETE事件）后再执行第131行的App。当然，那样就会在状态机中引入新的状态，代码就比较复杂了。因此，为了简单起见，我们在第130行使用了一个sleep函数暂停2秒。当然该行默认是注释掉的，仅在异步模式下才需要打开，同样的情况还有第10行。

至此我们的例子就结束了。从本例子中可以看出，使用C语言进行IVR编程也不是很复杂的，虽然代码行数与Lua相比大约增加了一倍。

19.3 用ESL写一个ACD

在第12章中，我们讲了使用mod_fifo和mod_callcenter做呼叫中心的例子，但在实际

⊖ 注意，这里说的打断是指在ESL程序中打断，我们使用play_and_get_digits在任何时候都是可以使用DTMF按键打断当前播放的提示音的。

使用中，还是有大量的用户会自己写业务逻辑进行座席分配（ACD）。既然我们已经学会了ESL，也不妨试着自己写一个。

在此我们还是使用Outbound模式来实现ACD。实现思路是——以来话用户为驱动，当有用户呼入时，查找是否有空闲座席，如果有，则转接到相应的座席上；如果没有，则进行忙等待，不断重试，直到找到一个空闲的座席或者用户挂机。

我们还是先从main函数开始看。首先，在第131行，它初始化了一个MUTEX，该变量是一个全局变量，用于多线程间的互斥（在第17行定义，我们后面还会讲到）；第132行，初始化一些座席，我们也将在后面讲到；第135行，进入监听，有呼入请求即在新线程中执行acd_callback。

```
129 int main(void)
130 {
131     esl_mutex_create(&MUTEX);
132     init_agents();

133     esl_global_set_default_logger(ESL_LOG_LEVEL_INFO);
134     esl_log(ESL_LOG_INFO, "ACD Server listening at localhost:8040 ...\n");
135     esl_listen_threaded("localhost", 8040, acd_callback, NULL, 100000);

136     esl_mutex_destroy(&MUTEX);

137     return 0;
138 }
```

下面是我们的acd_callback函数。当有用户呼入时，首先在第64行和65行取出主叫用户的名称和主叫号码，并于第66行打印一条日志。在第67行，我们使用myevents命令通知FreeSWITCH我们希望收到与本Channel相关的所有事件。第74行设置的这个变量比较重要，它用于保证在呼叫座席失败时不挂断电话，同时我们也需要第75行，它用于在成功bridge后（即成功接通座席后），如果座席首先挂机，则挂断来话。在播放完欢迎音（第76行）后，我们将持续播放音乐（第77行），直到转到座席上。

```
57  static void acd_callback(esl_socket_t server_sock,
            esl_socket_t client_sock, struct sockaddr_in *addr)
58  {
59      esl_handle_t handle = {{ 0 }};
60      esl_status_t status = ESL_SUCCESS;
61      agent_t *agent = NULL;
62      const char *cid_name, *cid_number;

63      esl_attach_handle(&handle, client_sock, addr);

64      cid_name = esl_event_get_header(handle.info_event, "Caller-Caller-ID-Name");
65      cid_number = esl_event_get_header(handle.info_event, "Caller-Caller-ID-Number");
66      esl_log(ESL_LOG_INFO, "New Call From \"%s\" <%s>\n", cid_name, cid_number);

67      esl_send_recv(&handle, "myevents");
68      esl_log(ESL_LOG_INFO, "%s\n", handle.last_sr_reply);
69      esl_send_recv(&handle, "linger 5");
70      esl_log(ESL_LOG_INFO, "%s\n", handle.last_sr_reply);
```

```
71      esl_execute(&handle, "answer", NULL, NULL);
72      esl_execute(&handle, "set", "tts_engine=tts_commandline", NULL);
73      esl_execute(&handle, "set", "tts_voice=Ting-Ting", NULL);
74      esl_execute(&handle, "set", "continue_on_fail=true", NULL);
75      esl_execute(&handle, "set", "hangup_after_bridge=true", NULL);
76      esl_execute(&handle, "speak", " 您好，欢迎致电，电话接通中，请稍候 ", NULL);
77      sleep(5);
78      esl_execute(&handle, "playback", "local_stream://moh", NULL);
```

第 79 行是一个 while 循环。我们将于第 82 行使用 esl_recv_timed 接收事件，如果在 1000 毫秒（即 1 秒）内收不到事件，则该函数返回 ESL_BREAK，这样就保证了我们的循环体不会因为收不到事件而阻塞。第 83 行判断如果在 1 秒内没有收到任何事件（如果收到事件我们则优先处理事件），并且没有找到对该呼入用户进行服务的座席，我们就在第 85 行查找空闲座席。第 86 行，如果找到了一个空闲坐席，则在第 89 行使用 break App 中断当前正在播放的音乐（这种方式只能在 async 模式下才有效），并于第 90 行使用 bridge 去桥接找到的座席。当然，如果本次没找到空闲的座席，则在下一秒时还有机会查找，因此只要呼入的用户不挂机，当有座席闲下来时，总能找到一个座席。

```
79      while(status == ESL_SUCCESS || status == ESL_BREAK) {
80          const char *type;
81          const char *application;

82          status = esl_recv_timed(&handle, 1000);

83          if (status == ESL_BREAK) {
84              if (!agent) {
85                  agent = find_available_agent();

86                  if (agent) {
87                      char dial_string[1024];
88                      sprintf(dial_string, "user/%s", agent->exten);
89                      esl_execute(&handle, "break", NULL, NULL);
90                      esl_execute(&handle, "bridge", dial_string, NULL);
91                      esl_log(ESL_LOG_INFO, "Calling: %s\n", dial_string);
92                  }
93              }
94              continue;
95          }
```

下面是事件处理程序，如果收到 CHANNEL_BRIDGE 事件，则说明已将来话与座席接通，这时我们要记住座席的 UUID（第 101 行），并打印一条日志。如果收到 CHANNEL_HANGUP_COMPLETE，则将还原当前座席的状态（置闲，第 106 行），并于第 107 行直接使用 goto 语句跳转到第 125 行的 end 标签，结束本次循环和本服务线程。

```
96          type = esl_event_get_header(handle.last_event, "content-type");

97          if (type && !strcasecmp(type, "text/event-plain")) {
98              /* esl_log(ESL_LOG_INFO, "Event: %s\n",
                esl_event_get_header(handle.last_ievent, "Event-Name")); */

99              switch (handle.last_ievent->event_id) {
```

```
100                 case ESL_EVENT_CHANNEL_BRIDGE:
101                     set_string(agent->uuid, esl_event_get_header(handle.
                           last_ievent, "Other-Leg-Unique-ID"));
102                     esl_log(ESL_LOG_INFO, "bridged to %s\n", agent->exten);
103                     break;
104                 case ESL_EVENT_CHANNEL_HANGUP_COMPLETE:
105                     esl_log(ESL_LOG_INFO, "Caller \"%s\" <%s> Hangup \n",
                        cid_name, cid_number);
106                     if (agent) reset_agent(agent);
107                     goto end;
```

如果收到 CHANNEL_EXECUTE_COMPLETE 事件，则说明有一个 App 执行完了，如果该 App 是 bridge 的话（第 110 行），则该事件中可以找到 variable_originate_disposition 包含了 bridge 的结果，我们先把它取出放到 disposition 变量中（第 111 行）。如果 disposition 为 CALL_REJECTED（第 113 行，拒接）或 USER_BUSY（第 114 行，被叫忙），则说明呼叫座席失败[⊖]，我们就于第 115 行重置该座席的状态，并于 116 行将当前座席设为 NULL，以后在下次事件循环时可以有机会重新选择座席。

总之，到此为止，主要的业务逻辑都已经实现了。如果成功接通座席，则等待双方挂机后退出；如果未接通座席，则尝试重新查找座席。

```
108                 case ESL_EVENT_CHANNEL_EXECUTE_COMPLETE:
109                     application = esl_event_get_header(handle.last_
                           ievent, "Application");
110                     if (!strcmp(application, "bridge")) {
111                         const char *disposition = esl_event_get_
                            header(handle.last_ievent,
                            "variable_originate_disposition");
112                         esl_log(ESL_LOG_INFO, "Disposition: %s\n", disposition);
113                         if (!strcmp(disposition, "CALL_REJECTED") ||
114                             !strcmp(disposition, "USER_BUSY")) {
115                             reset_agent(agent);
116                             agent = NULL;
117                         }
118                     }
119                     break;
120                 default:
121                     break;
122             }

123         }
124     }
125 end:
126     esl_log(ESL_LOG_INFO, "Disconnected! status = %d\n", status);
127     esl_disconnect(&handle);
128 }
```

到这里，让我们再回到程序的开始处。第 6 行定义了座席的状态，包括 IDLE（空闲）和 BUSY（忙），其他状态未用，只是作为一个例子，在实际的应用中座席会有更多的状

⊖ 如果呼叫成功，disposition 的值将是 SUCCESS。当然你可以尝试修改第 113 ~ 114 行的判断条件，把所有非 SUCCESS 的值都认为是失败。

态，如签入、签出、话后处理等。第 11 行定义了座席的结构体类型，它包括座席的分机号（exten）、通话中的 UUID 和座席当前的状态。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <esl.h>

4  #define MAX_AGENTS 3
5  #define set_string(dest, str) strncpy(dest, str, sizeof(dest) - 1)

6  typedef enum agent_state_s {
7      AGENT_IDLE,
8      AGENT_BUSY,
9      AGENT_FAIL
10 } agent_state_t;

11 typedef struct agent_s {
12     char exten[10];
13     char uuid[37];
14     agent_state_t state;
15 } agent_t;
```

出于简单起见，我们定义了几个全局变量。第 16 行的座席数组用于存放座席，目前我们最大可以有 3 个座席（由第 4 行的宏限制）。第 17 行定义了一个 MUTEX（互斥）⊖，由于我们是多线程的处理程序，当有多个电话同时呼入同时需要查找座席时，它用于保护临界区。第 18 行定义 last_agent_index 变量用于记录最后查询过的座席的位置，以支持轮循（Round Robin）的方式查找座席。在此，我们将它初始化为座席数组的最大值，以后它再启动会从第一个座席开始查。

```
16 static agent_t AGENTS[MAX_AGENTS] = { { 0 } };
17 static esl_mutex_t *MUTEX;
18 static int last_agent_index = MAX_AGENTS - 1;
```

下面是初始化座席的函数，我们这里没有签入、签出等操作，因此所有的初始化操作都是硬编码的。在此我们初始化了三个座席：1000 ~ 1002。

```
19 void init_agents()
20 {
21     set_string(AGENTS[0].exten, "1000");
22     set_string(AGENTS[1].exten, "1001");
23     set_string(AGENTS[2].exten, "1002");
24 }
```

下面我们实现了一个简单的查找空闲坐席的算法。第 29 行首先锁定临界区，以防止其他线程同时查找（其他线程将在临界区外等待）。第 31 行我们使用一个无限循环，从上一次查找过的座席的下一个开始查起（第 32 ~ 37 行保证 agent 指针永远指向下一个座席），如果找到空闲座席的话（第 40 行），则将当前座席置忙（第 41 行），并将当前座席的指针返回给

⊖ esl_mutex_t 是 ESL 库中的一个简单的多线程互斥的实现，它实际上是对操作系统底层的实现的一个跨平的的封装，如在有 pthread 库的平台上使用 pthread_mutex_t 实现，在 Windows 平台上则使用 CRITICAL_SECTION 实现。

调用者（第 43 行）。注意，在这里第 42 行是非常关键的，它将执行解锁操作，如果在返回之前忘记对临界区解锁的话，将会造成死锁。该函数在找不到空闲座席的情况下将会返回 NULL（第 48 行）。

```
25 agent_t *find_available_agent()
26 {
27     int last;
28     agent_t *agent;

29     esl_mutex_lock(MUTEX);
30     last = last_agent_index;

31     while (1) {
32         if (last_agent_index >= MAX_AGENTS - 1) {
33             last_agent_index = 0;
34         } else {
35             last_agent_index++;
36         }

37         agent = &AGENTS[last_agent_index];

38         esl_log(ESL_LOG_INFO, "Comparing agent [%d:%s:%s]\n",
39             last_agent_index, agent->exten, agent->state == AGENT_IDLE ?
                 "IDLE" : "BUSY");

40         if (agent->state == AGENT_IDLE) {
41             agent->state = AGENT_BUSY;
42             esl_mutex_unlock(MUTEX);
43             return agent;
44         }

45         if (last_agent_index == last) break;
46     }

47     esl_mutex_unlock(MUTEX);
48     return NULL;
49 }
```

下面函数用于把用完的座席还原，以便接听下一个电话（在下一次查找空闲座席时有机会找到它）。

```
50 reset_agent(agent_t *agent)
51 {
52     esl_mutex_lock(MUTEX);
53     agent->state = AGENT_IDLE;
54     *agent->uuid = '\0';
55     esl_mutex_unlock(MUTEX);
56 }
```

至此，我们的 ACD 服务程序就写好了。编译后执行一下，日志输出如下。其中，第 3 行显示 1006 呼入。第 6 行找到空闲座席 1000。第 7 行显示正在呼叫该座席。第 8 行显示被拒接，然后继续找到 1001 并呼叫（第 9 ~ 10 行），再次被拒接（第 11 行）后最终于在第 14 行呼通 1002。座席挂机时第 15 行显示呼叫的结果为 SUCCESS。第 16 行主叫挂机。第 17

行 Socket 中断线程并退出。

```
$ ./acd
[INFO] acd.c:165 main() ACD Server listening at localhost:8040 ...
[INFO] acd.c:83 acd_callback() New Call From "1006" <1006>
[INFO] acd.c:86 acd_callback() +OK Events Enabled
[INFO] acd.c:88 acd_callback() +OK will linger 5 seconds
[INFO] acd.c:48 find_available_agent() Comparing agent [0:1000:IDLE]
[INFO] acd.c:114 acd_callback() Calling: user/1000
[INFO] acd.c:138 acd_callback() Disposition: CALL_REJECTED
[INFO] acd.c:48 find_available_agent() Comparing agent [1:1001:IDLE]
[INFO] acd.c:114 acd_callback() Calling: user/1001
[INFO] acd.c:138 acd_callback() Disposition: CALL_REJECTED
[INFO] acd.c:48 find_available_agent() Comparing agent [2:1002:IDLE]
[INFO] acd.c:114 acd_callback() Calling: user/1002
[INFO] acd.c:128 acd_callback() bridged to 1002
[INFO] acd.c:138 acd_callback() Disposition: SUCCESS
[INFO] acd.c:131 acd_callback() Caller "1006" <1006> Hangup
[INFO] acd.c:155 acd_callback() Disconnected! status = 0
```

在本例中，我们使用 ESL 的 Outbound 方式实现了一个简单的 ACD 程序。该程序必须使用 ESL 的 async 模式才能工作，即在 Dialplan 中需要进行如下设置：

```
<action application="socket" data="127.0.0.1:8040 async full"/>
```

另外，该程序与上一节的程序有许多相似之处，在细节上又有诸多不同，读者可以自己对比体会一下。

19.4 用 Inbound 模式实现 IVR

上面讲到的例子大都是基于 Outbound 模式的，实际上 Inbound 模式也能完成同样的事情。下面我们仍以空中充值服务为例，来讲一下如何使用 Inbound 模式。这一次，我们把程序命名为 icharge。

与 Outbound 模式不同，Inbound 模式只跟 FreeSWITCH 建立一个 Socket 连接，即使有多路电话进来，所有的事件和控制命令也共享同一个 Socket 连接。当我们的服务程序接收到事件后，可以启用单线程处理，也可以启用多线程处理，但无论如何，Socket 连接只有一个。两者的另一个不同之处我们在上一章也讲过，那就是 Outbound 模式的 ESL 程序是一个服务器，在逻辑上作为通话的一方参与到通话中；而 Inbound 式的 ESL 程序相对于 FreeSWITCH 来讲相当于一个客户端，它主动连接到 FreeSWITCH 上收取事件并进行命令控制，相当于一个第三者控制通话。那么，为了让这个第三者能感知的来话信息，我们可以使用如下 Dialplan：

```
<extension name="Charge">
  <condition field="destination_number" expression="^1234$">
    <action application="set" data="service=icharge"/>
    <action application="park" data=""/>
  </condition>
</extension>
```

上述 Dialplan 的设计思路是：当有人拨打 1234 使用我们的 icharge 空中充值服务时，我们事先用 set App 设置一个 service 通道变量，然后将电话置为 park 状态。这样我们的 icharge 程序就可以在有电话进来时收到 CHANNEL_PARK 事件，并检查这里的 service 类型进行相应的处理。

通过前面的介绍，笔者想大家应该已经见识过了，C 语言的代码比较冗长，因此，在这里我们不再实现完整的查询和充值服务，而只实现一个相对简单的查询功能，以节省篇幅。剩下的部分感兴趣的读者可以自己练习一下。

好了，我们这次将源代码命名为 icharge.c，还是从 main 函数开始看。其中，我们于第 74 行通过 esl_connect 函数主动连接到 FreeSWITCH 上，该连接以后就通过 handle 来标识；第 80 行，订阅我们需要的事件，这样 FreeSWITCH 就会将相关的事件发给我们。

```
68  int main(void)
69  {
70      esl_handle_t handle = {{ 0 }};
71      esl_status_t status;
72      const char *uuid;

73      esl_global_set_default_logger(ESL_LOG_LEVEL_INFO);

74      status = esl_connect(&handle, "127.0.0.1", 8021, NULL, "ClueCon");

75      if (status != ESL_SUCCESS) {
76          esl_log(ESL_LOG_INFO, "Connect Error: %d\n", status);
77          exit(1);
78      }

79      esl_log(ESL_LOG_INFO, "Connected to FreeSWITCH\n");
80      esl_events(&handle, ESL_EVENT_TYPE_PLAIN,
81          "CHANNEL_PARK CHANNEL_EXECUTE_COMPLETE CHANNEL_HANGUP_COMPLETE");
82      esl_log(ESL_LOG_INFO, "%s\n", handle.last_sr_reply);
```

第 83 行，我们设置本次连接的 handle 的 event_lock 成员变量为“1”，表示我们希望所有的 App 和事件都以同步的方式执行（类似于上面讲到的 Outbound 模式中的同步模式），这样我们需要处理的状态机就比较简单了。注意，这里为了简单仅实现了一个单线程的处理机制，因此不能使用 sleep 等浪费时间的函数，否则就会影响到其他的通话（我们用一个线程也可以处理几千个并发的呼叫）。

接着是一个 while 循环（第 84 行），它阻塞地接收事件，当有事件到来时，便使用 process_event 对事件进行处理。注意，由于 ESL 是双向的，有可能当程序正在向 FreeSWITCH 发送命令并等待 FreeSWITCH 返回结果时，正好收到一个事件，这时 FreeSWITCH 便会将收到的事件缓存到一个内部链表中，这里 esl_recv_event 的第二个参数为“1”，表示优先检查内部缓存链表，以防止错过事件。

```
83      handle.event_lock = 1;
84      while((status = esl_recv_event(&handle, 1, NULL)) == ESL_SUCCESS) {
90          if (handle.last_ievent) {
91              process_event(&handle, handle.last_ievent);
```

```
92          }
93      }
94      esl_disconnect(&handle);
95      return 0;
96  }
```

下面我们可以倒回到文件开始处。最开始的部分跟 19.1.3 节讲到的类似，没什么特别需要注意的。

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <esl.h>

04  #define ERROR_PROMPT "say:输入错误，请重新输入"

05  int check_account_password(const char *account, const char *password)
06  {
07      return (!strcmp(account, "1111")) && (!strcmp(password, "1111"));
08  }
```

主要的业务逻辑在 process_event 函数里。第 12 行，我们还是使用一个 switch 语句来判断收到的事件类型。如果收到 CHANNEL_PARK 事件（第 13 行），则我们首先检查 service 通道变量，如果它不存在或不等于 icharge，说明它不是我们关心的业务类型，我们直接用 break 跳过后续的处理。否则，继续往下进行打印日志（第 20 行）、应答（第 21 行）、播放欢迎词（第 24 行）等。

```
09  void process_event(esl_handle_t *handle, esl_event_t *event)
10  {
11      const char *uuid;

12      switch (event->event_id) {
13          case ESL_EVENT_CHANNEL_PARK:
14          {
15              const char *service;

16              service = esl_event_get_header(event, "variable_service");
17              esl_log(ESL_LOG_INFO, "Service: %s\n", service);

18              if (!service || (service && strcmp(service, "icharge"))) break;

19              uuid = esl_event_get_header(event, "Caller-Unique-ID");
20              esl_log(ESL_LOG_INFO, "New Call %s\n", uuid);

21              esl_execute(handle, "answer", NULL, uuid);
22              esl_execute(handle, "set", "tts_engine=tts_commandline", uuid);
23              esl_execute(handle, "set", "tts_voice=Ting-Ting", uuid);
24              esl_execute(handle, "speak", "您好", uuid);
25              /* esl_execute(&handle, "speak", "您好，欢迎使用空中充值服务", uuid); */
```

由于我们使用了 handle->event_block，所以我们后面执行的 App 不会抢占前面的 App，因而我们可以在这一步尽量多执行一些 App（注意，App 的执行在 FreeSWITCH 中是阻塞的，但在 C 语言这一端不是，所以不用担心这里会阻塞接收事件的线程）。

在此，我们没有在 C 语言中实现状态机，而是借用通道变量实现了一个简单的状态机。在第 27 行，我们设置通道变量 charge_state 的值为 WAIT_ACCOUNT，即等待输入账号。当账号输入完毕后，再在第 31 行将该变量的值设为 WAIT_PASSWORD，即等待输入密码。其中第 28 行和第 32 行分别实现询问账号和密码，这些语句在 C 语言这一端很快就执行完成了，但在 FreeSWITCH 中，会一个接一个地执行。在此过程中，我们将会收到一个又一个的 CHANNEL_EXECUTE_COMPLETE 事件。不过，并不是所有事件都进行处理，直到收到的事件中满足第 44 ~ 45 行的检查条件才处理。

```
26 again:
27             esl_execute(handle, "set", "charge_state=WAIT_ACCOUNT", uuid);

28             esl_execute(handle, "play_and_get_digits",
29                 "4 5 3 5000 # 'say:请输入您的账号，以井号结束' "
30                 ERROR_PROMPT " charge_account ^\\d{4}$", uuid);

31             esl_execute(handle, "set", "charge_state=WAIT_PASSWORD", uuid);

32             esl_execute(handle, "play_and_get_digits",
33                 "4 5 3 5000 # 'say:请输入您的密码，以井号结束' "
34                 ERROR_PROMPT " charge_password ^\\d{4}$", uuid);

35             break;
36         }
37         case ESL_EVENT_CHANNEL_EXECUTE_COMPLETE:
38         {
39             const char *application;
40             const char *charge_state;

41             uuid = esl_event_get_header(event, "Caller-Unique-ID");
42             application = esl_event_get_header(event, "Application");
43             charge_state = esl_event_get_header(event, "variable_charge_state");

44             if (!strcmp(application, "play_and_get_digits") &&
45                 !strcmp(charge_state, "WAIT_PASSWORD")) {
```

第 44 ~ 45 行的检查条件是：收到 CHANNEL_EXECUTE_COMPLETE，最后执行的 App 是 play_and_get_digits，当前的状态是 WAIT_PASSWORD。只有满足了这三个条件，我们才知道用户输入密码完成了。然后，我们在第 46 行和第 47 行分别取出用户输入的用户名和密码。如果用户名和密码检查（第 48 行）通过，我们就向用户播放当前的余额，然后挂机（第 50 ~ 52 行）。当然如果用户输入有误，我们也友好地告诉他（第 54 行），并使用 goto 语句（第 55 行）跳到第 26 行的 again 标签，让用户重新输入。

```
46                 const char *account =
                       esl_event_get_header(event, "variable_charge_account");
47                 const char *password =
                       esl_event_get_header(event, "variable_charge_password");

48                 if (account && password && check_account_password(account, password)) {
49                     esl_log(ESL_LOG_INFO, "Account: %s Balance: 100\n", account);
50                     esl_execute(handle, "speak", "您的余额是100元", uuid);
```

```
51                  esl_execute(handle, "speak", "再见", uuid);
52                  esl_execute(handle, "hangup", NULL, uuid);
53              } else {
54                  esl_execute(handle, "speak", "账号密码错误", uuid);
55                  goto again;
56              }
57          }
58          break;
59      }
```

当收到用户挂机事件时（第 60 行），我们除了打印一条日志外（第 62 行）没有什么要特别处理的。

```
60          case ESL_EVENT_CHANNEL_HANGUP_COMPLETE:
61              uuid = esl_event_get_header(event, "Caller-Unique-ID");
62              esl_log(ESL_LOG_INFO, "Hangup %s\n", uuid);
63              break;
64          default:
65              break;
66      }
67  }
```

至此，我们的 icharge 服务已准备完毕，编译运行后相关日志如下，读者可以对着源代码自行分析一下（当然，为了消除空行对行号的影响，日志中还是手工调整了行号）。

```
$ ./icharge
[INFO] icharge.c:79 main() Connected to FreeSWITCH
[INFO] icharge.c:80 main() +OK event listener enabled plain
[INFO] icharge.c:17 process_event() Service: icharge
[INFO] icharge.c:20 process_event() New Call 65aca71e-4ad0-42d9-a638-e25e7bd75890
[INFO] icharge.c:49 process_event() Account: 1111 Balance: 100
[INFO] icharge.c:62 process_event() Hangup 65aca71e-4ad0-42d9-a638-e25e7bd75890
```

在本例中，我们使用 Inbound 模式的 ESL 程序实现了我们的空中充值服务，并使用单线程处理多路通话。由于每一个呼入的 Channel 都有唯一的 UUID，因此，对各路 Channel 的控制也互不影响。读者可能已经注意到了，在 Outbound 模式中，我们使用的 esl_execute 函数的最后一个参数值是 NULL，而在 Inbound 模式中，我们指定了本路通话的 UUID。原因就是，在 Outbound 模式中，是多个 Socket 连接，每个 Channel 对应一个 Socket 连接，因而不需要明确指定 UUID（当然如果指定了也没错）；而在 Inbound 模式中，对所有的 Channel 的控制都是在同一个 Socket 上进行的，因而需要明确指定 Channel 的 UUID。在我们这个简单的应用中，UUID 可以动态地从事件中获取，因而我们不需要复杂的数据结构来保存这些 UUID。

另外，读者可能也注意到了，我们在使用 play_and_get_digits 取得用户输入的账户和密码时，分别指定了 charge_account 和 charge_password 变量，这样就可以在后面一起取出（如第 46 行和 47 行），即可以通过这种方法把变量的值都保存到通道变量中，而不需要再在 C 语言中建立相关的数据结构来存储中间结果。

19.5 使用 Java 连接 ESL

前面我们讲的都是 C 语言的例子，C 语言是比较基础的语言，因而比较容易说明问题。然而，毕竟 C 语言比较底层，现在大多数的应用系统都是 Java、Ruby、PHP、C# 等高级语言开发的，因而更多的需要使用这些语言直接访问 ESL。考虑到 Java 语言在企业应用开发中还是比较流行的语言，因此，本节我们一起来看一下它是如何跟 FreeSWITCH 交互的。

下面我们假定读者已经非常熟悉 Java，并在主机上安装好了 Java 环境，对于不熟悉的读者，也可以通过该例子开拓一下视野，但对于基础的 Java 知识我们就不详细讲了。

FreeSWITCH 的 ESL 通过 Swig 支持多语言，并使用 JNI 集成 C 与 Java 的接口。Java 的 ESL 库是默认不编译的，如果要使用 Java ESL，则需要手工进行编译。为了能让 FreeSWITCH 的编译脚本能找到你的 Java 安装路径，需要在 configure 阶段指定，如笔者主机上的 Java 路径是 /opt/jdk1.7.0_45，则笔者在进行 configure 时按如下的方法指定 Java 路径：

```
./configure --with-java=/opt/jdk1.7.0_45
```

如果 FreeSWITCH 以前已经编译过，也可以不用再次运行 configure，而是直接修改 libs/esl/java 目录下的 Makefile，将 LOCAL_CFLAGS 中相关的路径修改为 Java 的安装目录。比如，笔者修改后的 Makefile 中相应的行是这样的：

```
LOCAL_CFLAGS=-I../src/include -I/opt/jdk1.7.0_45/include -I/opt/jdk1.7.0_45/
include/linux -fPIC
```

修改完成后直接在 libs/esl/java 目录下执行 make 或在 libs/esl 目录下执行 make javamod 命令完成编译。编译完成后，会在 java 目录下生成 libesljni.so 和 esl.jar，我们马上就会用到它们。

下面我们就创建一个 Java 项目，写一个简单的测试的类 MyESLTest，并保存到文件 MyESLTest.java 中。该类的代码很简单，首先第一行导入 Java ESL 库中一类和函数；第 4 行在 main 函数中初始化一个事件变量（event）；在第 5 行装入 ESL 的 JNI 接口库（在此我们使用了绝对路径，如果将该库文件放入 Java 的库文件目录中，也可以使用相对路径）；第 6 行初始化一个连接 conn，通过 Inbound 模式连接到 FreeSWITCH 上；第 8 行订阅全部事件；第 10 行在 while 循环中接收一个事件；第 11 行打印出事件的名字；当然也可以把第 12 行的注释去掉，让第 11 行打印整个事件。

```
01  import org.freeswitch.esl.*;

02  public class MyESLTest {

03          public static void main(String [] args) {
04                  ESLevent event;

05                  System.load("/root/java/libesljni.so");
06                  ESLconnection conn = new ESLconnection("127.0.0.1",8021,"ClueCon");
```

```
                if (conn.connected() == 1) System.out.println("Connected");

                conn.events("plain","ALL");

                while (conn.connected() == 1) {
                        event = conn.recvEvent();

                        System.out.println(event.getHeader("Event-Name", -1));
                        //System.out.println(event.serialize("plain"));
                }

        }
}
```

在 Java 中使用的函数与 C 语言中都是类似的，只是命名和调用习惯不一样。在开发过程中可以参考 libs/esl/java/org/freeswitch/esl/*.java 文件中的接口定义。

Java 源文件需要经过编译才能执行，如果你使用 Java 的集成开发环境，你需要引入 esl.jar 并进行编译，不过在此我们简单地写一个 Makefile，然后使用 make 命令编译就可以了，Makefile 内容如下：

```
all:
        javac -cp esl.jar MyESLTest.java

clean:
        rm MyESLTest.class
```

编译完成后，将得到 MyESLTest.class 文件，然后，就可以使用下面的命令执行它了：

```
java -cp .:esl.jar MyESLTest
```

下面是该程序在笔者服务器上的输出结果：

```
root@server:~/java# java -cp .:esl.jar MyESLTest
Connected
HEARTBEAT
RE_SCHEDULE
RE_SCHEDULE
```

当然，该程序除了打印一些事件外什么也做不了，不过有了这个例子，相信很容易将 FreeSWITCH 集成到你现有的 Java 环境中了。更多的信息可以参考：http://wiki.freeswitch.org/wiki/Java_ESL。

除了标准的 ESL 库以外，还有一个基于异步 IO 的 Netty 框架的 ESL 库，有兴趣的读者，尤其是正在使用 Netty 框架进行开发的读者也可以研究一下：http://wiki.freeswitch.org/wiki/Java_ESL_Client。

19.6 使用 Erlang 控制呼叫流程

Erlang ⊖是一门函数式编程语言。相比其他语言来说，它比较小众，但有着 20 多年历

⊖ 参见 http://www.erlang.org。

史的它一点也不年轻[1]。它具有轻量级多进程、高并发、热代码替换等电信级的特性，并非常易于和适合于创建集群应用（Cluster）。更重要的是，它就是为了编写电信应运而生的，在云计算这个词异常火热的今天，Erlang 编程语言更被赋予了新的意义。

在 FreeSWITCH 中有一个原生的模块 mod_erlang_event，它作为一个隐藏的 Erlang 节点[2]可以与任何 Erlang 节点通信。并且，它也实现了类似 ESL 中的 Inbound 和 Outbound 的通信模式。非常方便与其他 Erlang 系统集成。多年来，笔者一直从事 FreeSWITCH 以及与 Erlang 相关的开发，因此，本节的内容是必不可少的了。

19.6.1 准备工作

首先，对于对 Erlang 不熟悉的读者来说，我们简单补一补课。

在 Erlang 中，进程间通信靠消息和邮箱机制，一个进程可以给另外一个进程发消息，如下面的代码给一个进程（Pid）发送 message 消息（感叹号代码发送消息）:

```
Pid ! message
```

Erlang 天生支持集群，每个 Erlang 实例就相当于集群中的一个节点，每个节点有一个名字，可以是短名字（如 'name@localhost'），也可以是长名字（如 'name@freeswitch.org' 或 'name@192.168.1.2'）。短名字的节点只能和短名字的节点进行通信，长名字的节点也只能与长名字的节点通信。在下面的例子中，FreeSWITCH 节点的名字叫 'freeswitch@loalhost'，而我们实现的 Erlang 节点的名字则叫 'test@localhost'。

相互能通信的节点间可以互发消息，如使用如下代码可以给一个远程节点上的 Pid 进程发送消息：

```
{Pid, 'freeswitch@localhost'} ! hello
```

Erlang 在变量全部以大写字母或下划线开头，以小写字母开头的则是常量，在 Erlang 中称为原子（atom）。原子可以作为进程的名字，也可以作为节点的名字。反过来讲进程的名字和节点的名字（如果有的话）必须是原子。所有以单引号括起来的字符串都是原子（而不管字每是大写还是小写）。

关于 Erlang 的基础知识我们就讲这些，有了这些知识，Erlang 新手也能大体理解以下的内容了。当然如果读者有一定 Erlang 基础就更好了。

接下来，我们再准备一下 Erlang 的环境。

在 FreeSWITCH 中，与 Erlang 节点通信的模块是 mod_erlang_event，该模块在 FreeSWITCH 安装时默认是不安装的，安装该模块要首先应确认你的机器上已经安装好 Erlang 的开发环境。笔者习惯通过源代码安装（从官方网站上下载源代码，解压后使用如下命令安装：./configure && make && make install），如果通过系统提供的管理工具安装的话

---

[1] 参见 http://www.erlang.org/course/history.html。

[2] 即 Hidden Node，它是一个标准的节点，但别的节点看不见它（形象一点的比喻，严格来说如果使劲看也能看得见），如果别的节点知道它的名字，则可以跟它通信。

请安装 erlang-devel 或 erlang-dev 包。

确认安装好 Erlang 环境以后，在 FreeSWITCH 源代码目录中重新执行 ./configure，以在产生的 Makefile 中能找到相关的 Erlang 环境。然后在 FreeSWITCH 源代码目录中执行下列命令安装 mod_erlang_event：

```
# make mod_erlang_event-install
```

安装完成后，我们要对该模块进行配置。先检查配置文件，确认 conf/autoload_configs/erlang_event.conf.xml 中有以下三行（其他的行不动）：

```
<param name="nodename" value="freeswitch@localhost"/>
<param name="encoding" value="binary"/>
<param name="cookie" value="ClueCon"/>
```

其中第 1 行的 nodename 是为了强制限定 FreeSWITCH 节点的名字，如果不配置的话，FreeSWITCH 会自己为节点起一个最适合的名字，但我们发现它自己起的名字并不总是正确的，因此在这里为了防止引起其他可能的错误，在此人工指定一个名字（注意这里我们使用了短名字，如果你的 Erlang 节点在另一台机器上，你应该使用长名字）。第 2 行，我们使用二进制编码。默认的节点间通信使用文本编码，文本编码比二进制编码效率要低一些。第 3 行，设置该节点的 Cookie。注意要与 Erlang 节点的 Cookie 相同。

然后就可以在 FreeSWITCH 中加载该模块了，命令如下：

```
freeswitch> load mod_erlang_event
```

19.6.2 将来话交给 Erlang 处理

在此我们先讨论 Erlang 的 Outbound 模式。当有电话到来时，FreeSWITCH（可以看作一个客户端）会连接到你的 Erlang 程序节点（可以看作一个服务器）上，并把进来的电话控制权交给它。

在 Dialplan 中有两种设置方法：

```
<extension name="test">
    <condition field="destination_number" expression="^7777$">
        <action application="erlang" data ="ivr:start test@localhost"/>
    </condition>
</extension>

<extension name="test">
    <condition field="destination_number" expression="^7778$">
        <action application="erlang" data ="ivr:! test@localhost"/>
    </condition>
</extension>
```

我们把这两种方法分别称为“7777 法”和“7778 法”，下面对这两种方法进行介绍。

1. 7777 法

在 7777 法里，如果你呼叫 7777，FreeSWITCH 会首先将当前的 Channel 置于挂起（park）状态，并给你的 Erlang 程序（test@localhost 节点）发送一个 RPC 调用，调用的函数为

ivr:start(Ref)。其中，ivr:start 是在 Dialplan 中定义的，Ref 则是由 FreeSWITCH 端生成的针对该请求的唯一引用。在 Erlang 端，start/1 函数应该创建（spawn）一个新的进程，并且将新进程的 PID 返回给 FreeSWITCH，FreeSWITCH 收到该 PID 以后，便会将后续的所有与该 Channel 相关的事件都送给该进程。

Erlang 端的代码如下所示。当收到 FreeSWITCH 的远程调用时，它将产生一个新进程，并在新的进程中运行 loop() 函数，同时原来的进程会将新进程的 Pid（NewPid）返回给 FreeSWITCH。

```
start(Ref) ->
     NewPid = spawn(fun() -> loop() end),
     {Ref, NewPid}.
```

这是用 Erlang 控制呼叫的最简单的方法。任何时候来了一个呼叫，FreeSWITCH 就发起一个远端 RPC 调用给 Erlang，然后 Erlang 端启动一个新进程来控制该呼叫。由于 Erlang 的进程都是非常轻量级的，因而这种方式非常优雅。

我们将在后面再讨论 loop() 函数。

2. 7778 法

除了使用 RPC 生成新的进程外，还可以用另外一种方法产生新进程。如上面的 7778 法对应的 Dialplan 所示，与 7777 不同的是，其中的 ivr:start 中的 start 被换成了“！”。该方法需要你首先在 Erlang 端启动一个进程，该进程监听所有进来的请求。当 FreeSWITCH 把电话路由到 Erlang 节点时，根据“！”语法的规定，FreeSWITCH 会向 Erlang 发送一个 {getpid, ...} 消息，意思是说：我这里来了一路电话，告诉我哪个进程（对应一个 Pid）可以处理，我好把相应的事件发送给它。这种方法比上一种方法稍微复杂一些，但程序控制更加灵活。

如下面的代码，第一行的 start 函数必须事先执行，它在第 2 行注册一个有名字的进程（名字为 ivr），然后进入 wait() 无限循环。

```
01 start() ->
02     register(ivr, self()),
03     wait().
```

在 wait() 循环中，当收到一个 {get_pid, UUID, Ref, FSPid} 消息时，表示 FreeSWITCH 中来了一路通话，它便在第 7 行启动一个新的进程（新的进程将调用 loop() 进行服务），并在第 8 行把该新进程的 Pid（NewPid）发送给 FreeSWITCH（FSPid）。然后在第 10 行重新进入 wait() 等待后续的请求。

```
04 wait() ->
05     receive
06         {get_pid, UUID, Ref, FSPid} ->
07               NewPid = spawn(fun() -> loop() end),
08               FSPid ! {Ref, NewPid},
09               io:format("Main Pid: ~p Child Pid: ~p~n", [self(), NewPid]),
10               wait();
11          _Any ->
12               wait()
13     end.
```

19.6.3 用 Erlang 实现空中充值服务

下面我们看一下 Erlang 版的空中充值服务是什么样子的。在此我们将使用“7777 法”将呼叫转给 Erlang 处理。当有新呼叫到来时，新启动的进程将执行 loop，并会立即收到一个初始的接通事件，该事件类似我们在 C 语言的 ESL 中收到的 handle.info_event 中的事件，它在 Erlang 中的消息格式是 {event, [UUID | Event]} }，其中，UUID 为当前 Channel 的 UUID，Event 是一个 proplist，我们可以从中获取各种参数。

将我们的 Erlang 版的空中充值服务命名为 echarge，则在 Dialplan 中使用如下设置便可以将来话路由到我们的 Erlang 节点了。

```
<extension name="test">
  <condition field="destination_number" expression="^7777$">
    <action application="erlang" data ="echarge:start test@localhost"/>
  </condition>
</extension>
```

下面我们来看代码。Erlang 程序不像 C 语言那样需要前向声明，因而我们不必把被调用的函数放到最前面，故这一次我们终于可以从程序的第一行开始看了。

首先，第 1 行声明一个 Erlang 模块，它需要与文件名对应。第 2 行导出一个 start 函数，以便被 FreeSWITCH 远程调用。第 3 行定义了一个宏，表示 FreeSWITCH 所在的 Erlang 节点的名称。第 4 行的宏与 C 语言版本的类似。

```
01  -module(echarge).
02  -export([start/1]).

03  -define(FS_NODE, 'freeswitch@localhost').
04  -define(ERROR_PROMPT, "say: 输入错误，请重新输入").
```

我们的 start 函数很简单，它使用了标准的 7777 法启动一个新进程，并执行 loop() 函数。

```
05  start(Ref) ->
06      NewPid = spawn(fun() -> loop() end),
07      {Ref, NewPid}.
```

下面是我们的 loop() 函数。在第 9 行使用 receive 语句等待接收消息。第 10 行是它收到的第一个消息。收到消息后，它将打印该 Channel 的 UUID（第 11 行），然后调用 send_msg 函数执行应答（第 12 行），并设置 TTS 参数。这些步骤都完成后，再调用 send_lock_msg 函数播放欢迎词（第 15 行）。

在这里，send_msg 负责给 FreeSWITCH 发送一条 Erlang 消息，FreeSWITCH 收到消息后便会立即执行它指定的 App。send_lock_msg 与 C 语言版本的 ESL 中的 handle.event_locck=1 的情况相似，主要是为了能让 App 顺序阻塞地执行。接下来，在第 16 行将执行一个函数询问账号和密码，我们稍后再来看该函数，现在只需要知道它与 C 语言版本中的调用 play_and_get_digits 类似。

在 Erlang 中，所有的消息都是异步的，因而上面所描述的操作都很快执行完了。接下来，它在第 17 行再次执行递归地调用 loop() 函数，并使用 receive 再次接收消息。Erlang 也

支持尾递归，因而不用担心消耗栈空间等。

```
8  loop() ->
9       receive
10           {call, {event, [UUID | _Event]} } ->
11                io:format("New call ~s~n", [UUID]),
12                send_msg(UUID, answer, ""),
13                send_msg(UUID, set, "tts_engine=tts_commandline"),
14                send_msg(UUID, set, "tts_voice=Ting-Ting"),
15                send_lock_msg(UUID, speak, "您好，欢迎使用空中充值服务"),
16                ask_account_and_password(UUID),
17                loop();
```

后续的消息格式都是类似第 18 行的格式（以 call_event 开头），在收到消息后，执行 process_event() 函数来进行处理。如果收到 call_hangup 消息，则表示电话已挂机，打印一条日志然后就不再进行循环了（不再执行 loop()，该进程将会终止），否则在收到其他消息时（如第 21、第 23 行）继续循环。

```
18           {call_event, {event, [UUID | Event]} } ->
19                process_event(UUID, Event),
20                loop();

21           call_hangup -> io:format("Call hangup~n", []);

22           ok -> loop();

23           _X ->
24                io:format("Ignoring message ~p~n", [_X]),
25                loop()
26      end.
```

下面是我们询问账号和密码的函数，它还是类似 19.2 节所讲中的样子，在通道变量中设置当前充值流程的状态（charge_state），然后调用 play_and_get_digits App 收集信息。

```
27 ask_account_and_password(UUID) ->
28      send_lock_msg(UUID, set, "charge_state=WAIT_ACCOUNT"),
29      send_lock_msg(UUID, play_and_get_digits,
30           "4 5 3 5000 # 'say:请输入您的账号，以井号结束' "
31           ?ERROR_PROMPT " charge_account ^\\d{4}$"),
32      send_lock_msg(UUID, set, "charge_state=WAIT_PASSWORD"),
33      send_lock_msg(UUID, play_and_get_digits,
34           "4 5 3 5000 # 'say:请输入您的密码，以井号结束' "
35           ?ERROR_PROMPT " charge_password ^\\d{4}$").
```

大部分业务逻辑都在 process_event 函数中实现的，当收到来自 FreeSWITCH 的消息时，便执行该函数。Erlang 中的 case 语句类似 C 语言中的 switch，因此在第 41 行使用 case 来判断收到的消息类型，我们比较关心 CHANNEL_EXECUTE_COMPLETE（第 42 行），并且只有当满足第 43 行的条件时（当前状态为 WAIT_PASSWORD 并且当前的 App 为 play_and_get_digits），我们才知道账号和密码已收集完毕。在第 44 和 45 行我们分别取到它们的值，然后在第 46 行调用 check_account_password 函数检查输入是否合法。如果输入无误（第 47 行），则向用户播放当前余额（第 49 行），并挂机（第 51 行）。否则，提示输入错误（第 53

行），然后在第 54 行重新调用 ask_account_and_password 询问用户名和密码（Erlang 中没有 goto 语句，因而无法跳转，这也是我们把这一部分单独写成一个函数的原因）。

```
36 process_event(UUID, Event) ->
37      Name = proplists:get_value(<<"Event-Name">>, Event),
38      App = proplists:get_value(<<"Application">>, Event),
39      State = proplists:get_value(<<"variable_charge_state">>, Event),
40      io:format("Event: ~s, App: ~s State: ~s~n", [Name, App, State]),
41      case Name of
42           <<"CHANNEL_EXECUTE_COMPLETE">> when
43                State =:= <<"WAIT_PASSWORD">>, App =:= <<"play_and_get_digits">> ->
44                Account = proplists:get_value(<<"variable_charge_account">>, Event),
45                Password = proplists:get_value(<<"variable_charge_password">>, Event),
46                case check_account_password(Account, Password) of
47                     true ->
48                          io:format("Account ~s Balance: 100~n", [Account]),
49                          send_lock_msg(UUID, speak, "您的余额是 100 元"),
50                          send_lock_msg(UUID, speak, "再见"),
51                          send_lock_msg(UUID, hangup, "");
52                     false ->
53                          send_lock_msg(UUID, speak, "账号密码错误"),
54                          ask_account_and_password(UUID)
55                end,
56                loop();
57           _ -> ok
58      end.
```

下面的函数与 C 语言中的类似，只是语法不同而已，它会返回 true 或 false。

```
59 check_account_password(Account, Password) ->
60      Account =:= <<"1111">> andalso Password =:= <<"1111">>.
```

下面几个函数都是与 FreeSWITCH 交互的消息，第 62 行为向 FreeSWITCH 节点发送一条消息，消息的内容主要在 Headers 参数中指定。读者可以在下面的代码中看到 send_msg 和 send_lock_msg 的不同。

```
61 send_msg(UUID, Headers) ->
62      {sendmsg, ?FS_NODE} ! {sendmsg, binary_to_list(UUID), Headers}.

63 send_msg(UUID, App, Arg) ->
64      send_msg(UUID, [
65           {"call-command", "execute"},
66           {"execute-app-name", atom_to_list(App)},
67           {"execute-app-arg", Arg}
68      ]).

69 send_lock_msg(UUID, App, Arg) ->
70      send_msg(UUID, [
71           {"call-command", "execute"},
72           {"event-lock", "true"},
73           {"execute-app-name", atom_to_list(App)},
74           {"execute-app-arg", Arg}
75      ]).
```

在本例中，我们也是仅实现了空中充值服务中的查询部分。Erlang 的代码看起来也很直

观，而且我们没有依赖任何其他的库文件，完全是利用纯粹的 Erlang 消息与 FreeSWITCH 通信。将上述代码存为 echarge.erl，然后在 Makefile 中加入如下配置：

```
echarge: echarge.erl
    erlc echarge.erl

echarge-run:
    erl -PA . -setcookie ClueCon -sname test@localhost
```

然后执行 make echarge 就可以编译该文件了。编译完毕后，将产生 echarge.beam 文件。Erlang 的概念跟 Java 相似，它也是在一个虚拟机上运行的，这里编译生成的 .beam 文件相当于 Java 的 .class 文件，也是跨平台的。

然后，使用 make echarge-run 启动一个 Erlang 节点，该节点的名字是 test@localhost，以便 FreeSWITCH 能够在 Dialplan 中连接到它。用电话呼叫 7777，部分日志如下：

```
$ make echarge-run
erl -PA . -setcookie ClueCon -sname test@localhost
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8]...

Eshell V5.9.1  (abort with ^G)
(test@localhost)1> New call 4acb4e76-a7da-4b1b-bde0-0cc6921b4b77
Event: CHANNEL_PARK, App: undefined State: undefined
Event: CHANNEL_EXECUTE_COMPLETE, App: answer State: undefined
Event: CHANNEL_EXECUTE_COMPLETE, App: set State: WAIT_ACCOUNT
Event: CHANNEL_EXECUTE_COMPLETE, App: set State: WAIT_PASSWORD
Event: CHANNEL_HANGUP_COMPLETE, App: undefined State: WAIT_PASSWORD
Call hangup
```

19.6.4　用 Erlang 状态机实现空中充值服务

在上一节，我们使用了 Erlang 原始的消息收发机制控制 FreeSWITCH。在实际的 Erlang 使用中，大部都是都使用 OTP（Open Telecom Platform，开放电信平台）框架进行开发的。OTP 框架定义了一些标准的通用开发框架，使得 Erlang 项目开发更标准，也更简单。gen_fsm 就是其中的一个有限状态机（Finite State Machine）的实现。在本质上，IVR 服务最适合使用状态机实现，因为业务流程本来就是在各种不同状态中迁移的。

下面我们就来看一看如何利用 gen_fsm 使我们的空中充值服务的代码看起来更优雅。由于使用了 Erlang 标准的状态机，因此我们不再依赖于在 FreeSWITCH 中绑定通道变量的状态机实现，而且我们也不再使用 send_lock_msg 之类的阻塞执行过程，而是让所有过程都是异步的。下面你会看到这一节在 Erlang 中实现起来是多么的简单和优雅。

首先，我们将服务命名为 fsmcharge。代码第 2 行导出 gen_fsm 标准的函数。第 3 行导出所有的状态函数。我们还在第 7 行定义了一个 LOG 宏，用于输出带有行号的日志。

```
01  -module('fsmcharge').

02  -export([start/1, init/1, handle_info/3, terminate/3]).
03  -export([welcome/2, wait_account/2, wait_password/2,
04      wait_play_balance/2, wait_play_goodbye/2, wait_hangup/2]).
```

```
05 -define(FS_NODE, 'freeswitch@localhost').
06 -define(ERROR_PROMPT, "say: 输入错误，请重新输入").
07 -define(LOG(Fmt, Args), io:format("~b: " ++ Fmt ++ "~n", [?LINE | Args])).
```

下面我们定义一条记录（Erlang 中的 Record），其中 fsnode 为 FreeSWITCH 的节点的名称，其他的字符也都很直观了，不多解释。这些字段本身没什么用，我们实现该框架是为了方便以后扩充。

```
08 -record(state, {
09     fsnode          :: atom(),                  % FreeSWITCH node name
10     uuid            :: undefined | binary(),     % Channel uuid
11     cid_name        :: undefined | binary(),     % Caller id name
12     cid_number      :: undefined | binary()      % Caller id number
13 }).
```

第 14 行指出是在 start 函数中。在第 15 行调用了 gen_fsm:start() 产生了一个新进的 gen_fsm 进程，它将在新进程中回调第 17 行的 init 函数，可以在这里面进行一些初始化工作。init 函数的返回值在第 19 行，其中第 1 个值 ok 表示初始化成功；第 2 个值 welcome 表示下一个状态，如果后续收到消息，则调用 welcome 函数；第 3 个值 State 是一个内部状态变量，用于记录一些东西，在此，我们只是记住了 FreeSWITCH 的节点名（第 18 行）。

```
14 start(Ref) ->
15     {ok, NewPid} = gen_fsm:start(?MODULE, [], []),
16     {Ref, NewPid}.

17 init([]) ->
18     State = #state{fsnode = ?FS_NODE},
19     {ok, welcome, State}.
```

当第一个消息到来时，将回调 welcome 函数。第一个消息的格式如第 21 行粗体字部分所示。该在函数中，我们在第 22 ~ 23 行取出主叫的名称和号码，第 24 行写一条日志，然后应答，设置 TTS 参数等。在第 28 行发出播放欢迎词的消息后，即返回第 29 行的值。其中，第 29 行的第一个参数为 next_state，它指定下一个状态；下一个状态的值在第 2 个参数中，还是 welcome；第 3 个参数（在第 30 行）中的内部状态变量被更新了，填入了当前的主叫用户名称和号码。

```
20 %% The state machine
21 welcome({call, _Name, UUID, Event}, State) ->
22     CidName = proplists:get_value(<<"Caller-Caller-ID-Name">>, Event),
23     CidNumber = proplists:get_value(<<"Caller-Caller-ID-Number">>, Event),
24     ?LOG("New Call \"~s\" <~s>", [CidName, CidNumber]),
25     send_msg(UUID, answer, ""),
26     send_msg(UUID, set, "tts_engine=tts_commandline"),
27     send_msg(UUID, set, "tts_voice=Ting-Ting"),
28     send_msg(UUID, speak, "您好，欢迎使用空中充值服务"),
29     {next_state, welcome,
30         State#state{uuid=UUID, cid_name=CidName, cid_number=CidNumber}};
```

当有后续的消息到来时，都将会继续回调 welcome，但消息格式永远不会匹配到第 21 行。如果到来的消息匹配到第 31 行，那么说明有一个 App 执行完了。我们接着在第 32 行

进行判断，如果完成的 App 是 speak（第 33 行），则说明欢迎语已播放完毕，我们立即发送 play_and_get_digits App 以提示用户输入账号（第 34 行），这次我们的返回值在第 37 行，它的第 2 个参数变为 wait_account，即以后不管收到什么消息都会回调 wait_account 函数（也就是说我们进入了下一个状态）。另外，第 41 行会匹配任何消息，它的意思是，如果在这过程中收到的消息与第 21 和第 31 行都不匹配，就执行该行，它什么也不做，返回的下一个状态与当前状态相同，即不发生任何状态转换。

```
31  welcome({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, Event}, State) ->
32      case proplists:get_value(<<"Application">>, Event) of
33          <<"speak">> ->
34              send_msg(UUID, play_and_get_digits,
35                  "4 5 3 5000 # 'say:请输入您的账号，以并号结束' "
36                  ?ERROR_PROMPT " charge_account ^\\d{4}$"),
37              {next_state, wait_account, State};
38          _ ->
39              {next_state, welcome, State}
40      end;

41  welcome(_Any, State) -> {next_state, welcome, State}.
```

在 wait_account 状态下，我们仅会收到一个 CHANNEL_EXECUTE_COMPLETE 消息（第 42 行），就是第 34 行的执行结果，说明第 34 行的 App 已执行完毕，因而不需要再深入地去判断事件中的 Application 的值了，只需要继续发送下一个命令询问密码（第 43 行），并将下一个状态设为 wait_password（第 46 行）。第 47 行的意思是不管收到任何不期望的事件，都不会发生状态转换。

```
42  wait_account({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, _Event}, State) ->
43      send_msg(UUID, play_and_get_digits,
44          "4 5 3 5000 # 'say:请输入您的密码，以井号结束' "
45          ?ERROR_PROMPT " charge_password ^\\d{4}$"),
46      {next_state, wait_password, State};

47  wait_account(_Any, State) -> {next_state, wait_account, State}.
```

当用户输入密码完毕后，我们又收到一个 CHANNEL_EXECUTE_COMPLETE，程序执行到第 48 行。此时，我们便可以在第 51 行检查账号和密码的有效性了。如果用户输入错误（第 56 行，注意这里我们先说错误的情况），则播放错误提示（第 57 行），然后将下一个状态变为 welcome，下一次收到事件时将回到我们最开始的 welcome 状态，并回调 welcome 函数。

如果用户输入正确，则在第 54 行播放余额，并进入下一个状态 wait_play_balance（第 55 行）等待播放完成。

```
48  wait_password({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, Event}, State) ->
49      Account = proplists:get_value(<<"variable_charge_account">>, Event),
50      Password = proplists:get_value(<<"variable_charge_password">>, Event),

51      case check_account_password(Account, Password) of
52          true ->
```

```
53              ?LOG("Account: ~s, Balance: 100", [Account]),
54              send_msg(UUID, speak, "您的余额是 100 元"),
55              {next_state, wait_play_balance, State};
56          false ->
57              send_msg(UUID, speak, "账号密码错误"),
58              {next_state, welcome, State}
59      end;

60  wait_password(_Any, State) -> {next_state, wait_password, State}.
```

余额播放完毕后，回调第 61 行的函数，继续在第 62 行播放“再见”，并进入下一个状态 wait_play_goodbye。

```
61  wait_play_balance({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, _
Event}, State) ->
62      send_msg(UUID, speak, "再见"),
63      {next_state, wait_play_goodbye, State};

64  wait_play_balance(_Any, State) -> {next_state, wait_play_balance, State}.
```

当“再见”也播放完成后，我们的程序会执行到第 65 行，这时没有其他事情要做了，我们便于第 66 行发送 hangup 命令通知 FreeSWITCH 挂机。为了记录一些信息，我们并没有立即退出进程，而是继续将状态转移到下一个状态 wait_hangup 等待挂机消息（第 67 行）。

```
65  wait_play_goodbye({call_event, <<"CHANNEL_EXECUTE_COMPLETE">>, UUID, _
Event}, State) ->
66      send_msg(UUID, hangup, ""),
67      {next_state, wait_hangup, State};

68  wait_play_goodbye(_Any, State) -> {next_state, wait_play_goodbye, State}.
```

当收到 CHANNEL_HANGUP_COMPLETE，表示电话已经释放完成了。我们可以从该事件中取到呼叫时长（Duration）和计费时长（Billsec）等信息（第 70 ~ 71 行），打印出来（第 72 行）或（以后）写入数据库。当然，到此为止我们真的没有什么要留恋的了，因而这次的返回值中（第 73 行）第一个字段是 stop，表示不再进行状态转移了，而是要结束本进程。

```
69  wait_hangup({call_event, <<"CHANNEL_HANGUP_COMPLETE">>, _UUID, Event},
        State) ->
70      Duration = proplists:get_value(<<"variable_duration">>, Event),
71      Billsec = proplists:get_value(<<"variable_billsec">>, Event),
72      ?LOG("Call Ended, Duration: ~s Billsec: ~s", [Duration, Billsec]),
73      {stop, normal, State};

74  wait_hangup(_Any, State) ->
75      {next_state, wait_hangup, State}.
```

理论上讲不会执行到第 76 行，因为 call_hangup 消息是在 CHANNEL_HANGUP_COMPLETE 事件之后才发出发出的，而我们的进程已经终止了。但是，万一出现事件丢失的情况，多了这一层保障也是好的，我们还是会返回 stop（第 77 行）。

注意，这里的 call_hangup 消息是在 handle_info 函数中收到的。在 OTP 中，一般来说所有原始的 Erlang 消息都是在 handle_info 函数中处理的。我们与 FreeSWITCH 之间是节

点互联的关系，FreeSWITCH 并不知道我们使用的是 OTP 框架，仍然给我们发送原始的 Erlang 消息。

```
76  handle_info(call_hangup, _StateName, State) ->
77      {stop, normal, State};
```

所以，第 78 行就是 FreeSWITCH 发过来的所有消息的入口（除 call_hangup 之外）。在这里，我们收到消息后，取出事件的名字 EventName（第 79 行），并重新组合成一个新的消息（第 81 行，我们把 EventName 放到消息中有助于在函数级别进行匹配），作为参数回调当前的状态函数（StateName）。

当然，第 81 行的实现属于一种取巧的方法。实际上真正的 gen_fsm 习惯是不使用第 81 行（可以注释掉），而使用第 82 行的 gen_fsm:send_event 函数再向自己（self）发送一次消息，并于第 83 行返回下一个状态（即当前的状态不做任何变化），让 gen_fsm 机制去回调当前的状态函数。当然，那样的话会多一次消息收发的过程，不是很高效，因此我们把第 82 行和第 83 行注释掉了[⊖]。

```
78  handle_info({EventType, {event, [UUID | Event]}}, StateName, State) ->
79      EventName = proplists:get_value(<<"Event-Name">>, Event),
80      ?LOG("State: ~s Event: ~s", [StateName, EventName]),
81      StateName({EventType, EventName, UUID, Event}, State);
82      % gen_fsm:send_event(self(), {EventType, EventName, UUID, Event}),
83      % {next_state, StateName, State};

84  handle_info(_Any, StateName, State) ->
85      {next_state, StateName, State}.
```

最后，将进程终止时，还是有机会进行回调清理一个现场的。如果进程正确终止，则回调第 86 行的函数；否则将回调第 87 或第 91 行的函数。如果是回调第 87 行，则说明进程是异常终止的（可以从 _Reason 参数中取得异常原因），这时最好是通知 FreeSWITCH 端挂机（第 89 行）。

```
86  terminate(normal, _StateName, _State) -> ok;
87  terminate(_Reason, _StateName, #state{uuid = UUID}) when UUID =/= undefined ->
88      % do some clean up here
89      send_msg(UUID, hangup, ""),
90      ok;
91  terminate(_Reason, _StateName, #state{uuid = UUID}) -> ok;
```

其他的函数跟上一节讲过的一样，为了完整起见我们也列在这里。

```
92  check_account_password(Account, Password) ->
93      Account =:= <<"1111">> andalso Password =:= <<"1111">>.

94  send_msg(UUID, Headers) when is_list(Headers) ->
95      {sendmsg, ?FS_NODE} ! {sendmsg, binary_to_list(UUID), Headers}.

96  send_msg(UUID, App, Arg) ->
```

⊖ 这一点初学者不需要过多深入研究，知道这一点以后就应该对 OTP 完全了解了。而且，笔者在生产环境中使用的是自己修改过的 gen_fsfsm，这从根本上避免了多一次的消息发送，同时又符合 OTP 的使用习惯。

```
        send_msg(UUID, [
            {"call-command", "execute"},
            {"execute-app-name", atom_to_list(App)},
            {"execute-app-arg", Arg}
        ]).
```

下面是该程序的执行日志（有删节，并手工调整了行号）：

```
$ make fsmcharge-run
erl -PA . -setcookie ClueCon -sname test@localhost
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8] ...

Eshell V5.9.1  (abort with ^G)
(test@localhost)1>
80: State: welcome Event: CHANNEL_DATA
24: New Call "1006" <1006>
80: State: welcome Event: CHANNEL_PARK
80: State: welcome Event: CHANNEL_EXECUTE_COMPLETE
80: State: wait_account Event: CHANNEL_EXECUTE
80: State: wait_account Event: CHANNEL_EXECUTE_COMPLETE
80: State: wait_password Event: CHANNEL_EXECUTE
80: State: wait_password Event: DTMF
80: State: wait_password Event: CHANNEL_EXECUTE_COMPLETE
66: Account: 1111, Balance: 100
80: State: wait_hangup Event: CHANNEL_HANGUP_COMPLETE
72: Call Ended, Duration: 17 Billsec: 17
```

在本节的例子中，我们采用了完全的状态机实现了 IVR 应用。虽然代码有点长，但 Erlang 的状态机实现得很优雅，代码看起来很清晰，而在前面的例子中，如果 if…else 之类的判断多了，就会感觉很乱。这种状态机在其他语言中应该也不难实现。

19.6.5 其他

在上面的例子中，我们都使用了 Outbound 模式的 Erlang 连接。其实，该模块也支持 Inbound 模式。

使用 Inbound 模式连接 FreeSWITCH 很简单，首先我们启动一个 Erlang 节点。下列命令实现的就是启动一个 Erlang 节点并进入一个 Erlang Shell 等待我们输入命令：

```
$ erl -setcookie ClueCon -sname test@localhost
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:8:8] ...

Eshell V5.9.1  (abort with ^G)
(test@localhost)1>
```

接着输入给 FreeSWITCH 节点发送一个 register_event_handler 消息通知 FreeSWITCH 我们想在当前进程（Erlang Shell 进程）中收取事件：

```
(test@localhost)2> {event, 'freeswitch@localhost'} ! register_event_handler.
register_event_handler
```

这时在 FreeSWITCH 控制台上便能看到类似如下的 Debug 日志，提示有一个客户端节点 test@localhost 连接上来了：

```
[DEBUG] mod_erlang_event.c:2022 Launching listener, connection from node test@
localhost, ip 127.0.0.1
[DEBUG] mod_erlang_event.c:1031 Connection Open from 127.0.0.1
```

然后，我们再向其发送 {event, all} 消息订阅全部的事件：

```
(test@localhost)3> {event, 'freeswitch@localhost'} ! {event, all}.
{event, all}
```

FreeSWITCH 端的日志如下（这说明日志订阅成功了）：

```
[DEBUG] handle_msg.c:301 ALL events enabled
[DEBUG] handle_msg.c:314 enable event all
```

然后，在我们的 Erlang 节点上的 Shell 中输入 "flush()." 并按回车，就可以看到当前 Shell 收到的所有事件了，具体代码如下。其中，最初的两个 "ok" 消息是对我们上面发的两条消息的回复，如果这时候 FreeSWITCH 中产生了事件，我们也会一起看到。FreeSWITCH 每 20 秒产生一次 Heartbeat 事件，因此笔者恰好收到了一次（下面只显示了一部分）：

```
(test@localhost)7> flush().
Shell got ok
Shell got ok
Shell got {event,[undefined,
          {<<"Event-Name">>,<<"HEARTBEAT">>},
          {<<"Core-UUID">>,<<"238c20ac-54fb-4312-a6b4-8819fb8a10ac">>},
          {<<"FreeSWITCH-Hostname">>,<<"seven.local">>},
          {<<"FreeSWITCH-Switchname">>,<<"seven.local">>},
          {<<"FreeSWITCH-IPv4">>,<<"192.168.7.6">>},
```

如果有时间，可以多打几次 "flush()." 试试，应该能一直不停地收到新的事件。

当然，我们也可以直接向 FreeSWITCH 发送一个 API 命令，如：

```
(test@localhost)3> {api, 'freeswitch@localhost'} ! {api, version, ""}.
{api,version,[]}
```

再次输入 "flush()." 将会看到以下结果。可以看出，FreeSWITCH 除了返回 version 命令的结果外，还产生了一个 API 事件。

```
(test@localhost)4> flush().
Shell got {ok,<<"FreeSWITCH Version 1.5.8b+git~20131201T102929Z~558e484f05~64bit
(git 558e484 2013-12-01 10:29:29Z 64bit)\n">>}
Shell got {event,[undefined,
          {<<"Event-Name">>,<<"API">>},
          {<<"Core-UUID">>,<<"238c20ac-54fb-4312-a6b4-8819fb8a10ac">>},
          {<<"FreeSWITCH-Hostname">>,<<"seven.local">>},
          {<<"FreeSWITCH-Switchname">>,<<"seven.local">>},
          {<<"FreeSWITCH-IPv4">>,<<"192.168.7.6">>},
```

当然，上面是 mod_erlang_event 模块使用的原始消息格式。在该模块的源代码中，附带了一个 freeswitch.erl 文件，它包装了一些有用的函数，使用起来就方便多了。作为练习，我们可以将它复制到当前的目录，用以下命令编译它：

```
$ erlc freeswitch.erl
```

编译完成后将在当前目录下生成 freeswitch.beam，使用如下命令可以启动一个新的 Erlang 节点，并加载当前目录（由参数“-pa .”指定）下所有的 beam 文件：

```
$ erl -pa . -setcookie ClueCon -sname test@localhost
```

然后，就可以直接使用 freeswitch:api() 进行 API 命令调用了，如：

```
(test@localhost)1> freeswitch:api('freeswitch@localhost', version, "").
{ok,<<"FreeSWITCH Version 1.5.8b+git~20131201T102929Z~558e484f05~64bit
(git 558e484 2013-12-01 10:29:29Z 64bit)\n">>}

(test@localhost)2> freeswitch:api('freeswitch@localhost', show, "channels").
{ok,<<"\n0 total.\n">>}
```

更多的使用方法我们就不多解释了，读者可以自行参考一下 freeswitch.erl 文件中的注释以及该模块的 Wiki 页面：http://wiki.freeswitch.org/wiki/Mod_erlang_event。

19.7 定时呼叫

在本章的最后，我们再讲一个定时呼叫的例子。定时呼叫类似于现有 PSTN 网络中的叫醒服务，但在 FreeSWITCH 中我们肯定能玩出更多的花样。在此我们仅举一个最简单的例子。

在 UNIX 类的系统中，使用 crontab 作定时服务，我们可以用它定时执行一个脚本或程序。它的设置非常灵活，因此我们几乎可以用它作任何定时。

在 Shell 中使用 crontab -e 命令可以启动一个编辑器编辑当前用户的 crontab 文件，我们输入以下内容并存盘：

```
0 6 * * * /usr/local/freeswitch/bin/fs_cli -x "originate user/1006 &playback(/
tmp/wakeup.wav)"
```

以上内容中，以空格分隔的有多个域，前 5 个分别表示分钟、小时、月、日、星期，如果某个位置的值为“*”，则它可匹配任何值。如上述的定时就是每天早上的 6 点整执行后面的命令（如果是“0 * * * *”则表示每小时整点执行一次，“* * * * *”表示每分钟执行一次，以此类推）。

后面的命令实际上是使用 fs_cli 的 -x 参数执行一个 FreeSWITCH 的 API，它的操作很简单，即呼叫用户 1006，接通后播放声音文件 /tmp/wakeup.wav 叫用户起床。

如果想要执行的业务逻辑比较复杂（如未接听的情况下重呼，或得接听了但不按“1”就重呼），可以将逻辑放到一个 Lua 脚本中，用以下命令调用：

```
0 6 * * * /usr/local/freeswitch/bin/fs_cli -x "luarun /tmp/wakeup.lua"
```

具体的 Lua 脚本内容我们就不举例了。当然，如果觉得 Lua 不过瘾，也可以使用我们本章讲的 ESL 知识写一个 ESL 应用，如 /tmp/esl_wakeup，然后可以通过如下配置执行：

```
0 6 * * * /tmp/esl_wakeup
```

总之实现定时呼逻辑应该不是很复杂的事，如果有需要，读者可自行练习。Windows 系统上的用户也可以通过系统的定时任务定时执行程序或脚本，因为我们的实现方案是跨平台的。

19.8 小结

在本章，我们主要讲了使用 ESL 连接 FreeSWITCH 实现各种业务逻辑的例子。这些例子主要使用了 C 语言，因为 C 语言是一门基础语言，而且 FreeSWITCH 或 ESL 底层本身也是使用 C 语言实现的。在实现这些例子的过程中，我们还穿插着讲了一些 Makefile 以及系统编译的知识，所有的程序都是可以运行的，通过这些程序可方便读者进行实战演练。本章大部分例子都带有实际运行的结果和日志，以方便读者对照学习。

通过不同的例子，我们涵盖了 ESL 的 Inbound 及 Outbound 模式（以及同步及异步模式）的开发方法和实例，并尽量在不同的例子中使用不同的方法或调用不同的函数或参数，以尽量覆盖更多的内容。当然，由于篇幅限制，我们没有详细比较每一个细节的不同，这些需要读者多多实际练习和比较。

除了 C 语言外，我们还提供了 Java 和 Erlang 语言的例子，通过对这些语言的横向比较，读者可以对 ESL 的实现机制有一个更宏观的了解。理解了其中一种语言中的实现方法，就可对比理解其他语言。当然，再次由于篇幅限制，我们不可能覆盖所有编程语言，但希望这三种语言能覆盖大部分的用户。

在这些例子中，我们不厌其烦地、一遍又一遍地实现了我们的“空中充值服务”的例子，目的就是为了让大家在同一种语言的不同实现方式，以及不同语言的相同及不同实现方式下做一下横向的比较，以使读者更深入理解各种模式以及其优缺点，这样便能在以后使用时选择最好的实现方法。当然，不要在本章学了 ESL 就忘了前面的 Lua，把我们的充值服务也跟 Lua 版的比较一下，看自己到底喜欢哪一种。

在本章的最后我们还讲了一个定时呼叫的例子，该例子非常简单，但也经常有网友问到。当然，笔者提供该例子的真正目的是让大家换一下思路，克服思维定势，不要钻到 ESL 中就拔不出来了。原来该问题还可以用 crontab+shell 实现，原来还可以配合 Lua 脚本实现更强的逻辑，原来……肯定还有好多连笔者也没想到的好方法。

本章在我们主要针对 ESL 的同时，其实我们还讲了许多开发设计的思路及需要注意的问题以及诸如多线程、状态机、节点、集群、日志、源代码版本控制等各种方面的知识，希望这些对读者有所帮助。另外，也希望本章的这些知识对理解我们后面章节中将要讲到的 FreeSWITCH 的源代码阅读及基于 FreeSWITCH 的开发有所帮助。

第 20 章 源代码导读及编译指南

从本章开始，我们进入 FreeSWITCH 源代码的学习。使用开源项目的好处是我们可以阅读并修改其源代码。

有好多朋友在被问到如何阅读源代码时，都会告诉笔者，他们最大的困扰并不是看不懂代码，而是不知道从哪里下手，就好像是老虎吃天——无从下口。是的，FreeSWITCH 的源代码太长了，确实好像从哪里看都找不到源头。不过，也不要因此望而却步。在看比较大型的项目时，尤其是对着不熟悉的功能和代码时，总要经过这么一个过程。尤其是，有些读者在对 FreeSWITCH 本身还不熟悉的时候，就试图通过阅读源代码来了解系统的功能。笔者认为那是本末倒置的方法，不可取。而本书的读者，如果是从前面的章节一步一步读过来的，甚至做过了必要的实验，在实验的过程中遇到问题自己也尝试解决了，那么笔者相信大家已经掌握了 FreeSWITCH 的基本功能，接下来再去阅读源代码便很容易入手了。

当然，阅读源代码不仅是为了满足我们的好奇心——哪些功能是怎么实现的、系统还有何种没有公开（写到文档里）的功能等。更重要的是，如果我们熟悉了源代码，就可以修改它——不管是修复 BUG、增加功能并为开源项目做贡献，还是修改源代码以适合自己的需要，这些都能给我们带来很好的成就感。

接下来，我们从准备源代码环境开始，由浅入深地一步一步进入 FreeSWITCH 内部的神秘世界。

20.1 准备 FreeSWITCH 源代码环境

我们在第 3 章已经讲到多种获取源代码的方法。如果你需要基于 FreeSWITCH 源代码进行二次开发的话，笔者强烈建议学一点 Git 基本知识，并使用 Git 复制源代码。方法重述如下：

```
git clone git://git.freeswitch.org/freeswitch.git
```

要查看源代码，最好选择一个具有语法高亮功能的阅读器或编辑器。笔者在 Mac 平台上一般使用 Sublime Text 2，它是跨平台的，也可以在 Windows 上使用。当然也可以使用

一些经典工具，如在 UNIX 类系统上使用 vi/vim 或 Emacs，在 Windows 上可以使用 Visual Studio 等。关于在 Windows 平台中的编译方法我们在第 3 章已经讲过了，本章及后面章节的例子都是在 Mac 平台上完成的，它适用于大部分的 UNIX 类平台（如 Linux 等）。

20.2 FreeSWITCH 源代码目录结构

FreeSWITCH 的源代码目录中，src 目录中包含了绝大部分的源代码；libs 目录下是一些第三方的库和模块，如 libs/sofia-sip 就是 Nokia 的 SIP 库。

在 src 目录中，include 目录存放了系统大部分的头文件；不同模块的代码则分门别类地放到 mod 目录中不同的子目录中。系统的核心代码则直接在 src 目录中。

FreeSWITCH 模块的源代码（mod 目录）结构与图 5-1 所示的差不多，如表 20-1 所示。

表 20-1　FreeSWITCH 源代码模块目录结构

目　录	说　明
asr_tts	语音识别及合成相关模块
dialplans	Dialplan 模块
endpoints	Endpoint 模块，如 mod_sofia
formats	文件格式模块，如 mod_sndfile
legacy	旧模块，如基于 Lua 5.1 的 mod_lua
loggers	日志模块
sdk	一些例子和宏
xml_int	XML 相关的模块
applications	提供各种应用功能的模块，如 mod_dptools 和 mod_commands
directories	LDAP
event_handlers	事件处理模块
languages	嵌入式语言模块
say	不同语种的语言模块
timers	时钟和定时器模块

上面不同的分类也与 FreeSWITCH 内部模块的抽象大致对应，但也有例外的情况，比较典型的，比如一些模块可能由多个接口（Interface）实现，这样的模块会根据其主要功能放到对应的目录中，有时就直接放到 applications 目录中，相当于该目录中有一些多功能的模块。但随着时间的推移，某些单功能的模块也可能会被增加一些新的功能和接口（Interface），变成多功能模块。

20.3 FreeSWITCH 源代码导读

由于 FreeSWITCH 的代码非常多，为了节省篇幅，我们在本章中尽量不大段列出源代

码。读者在阅读本章时可以配合源代码进行。

FreeSWITCH 的更新速度比较快，因此，为了让读者能找到正确的行号，我们这里的源代码导读基于 Git 的 Tag v1.5.7，它是在本章写作是比较新的版本。读者可以使用下列命令切换到该 Tag：

```
git checkout v1.5.7
```

20.3.1 APR

FreeSWITCH 在设计之初就定位于跨平台的，它使用了跨平台较好的 APR 库[1]。APR 出身于 Apache[2]的代码。Apache 是网络上非常流行的 Web 服务器软件，其代码是公认的写得比较好的。在程序员这一行业，大家一致的观点是——要想提高开发水平，除了大量的练习外，还要大量地阅读其他优秀系统的源代码，而 Apache 就是常被推荐阅读的代码之一。

APR 的主要目的是为应用提供一个可移植的、平台无关的层。它的底层在不同平台上调用不同的库和函数来向上提供诸如文件系统访问、网络编程、进程和线程管理以及共享内存等一致的功能接口。使用 APR 开发的程序能够在所有被 Apache 所支持的平台上被干净地（最坏的情况也仅需要很小程度的修改）编译。

除了跨平台支持外，APR 的核心还提供系统内存、数据结构、线程、互斥锁等各种资源的管理和抽象等。大家都知道 C 语言的内存管理是非常令人头痛的，而 APR 通过内存池的管理大大提高了使用的方便性和安全性。

APR 实用库（APR-UTIL，或者 APU）是 APR 项目的另一个库。它在 APR 基础上使用统一标准的编程接口，提供了一部分功能函数集。APU 并不是在每一个平台上都有一个单独的模块，但是它为某些其他常用的资源一个类似的方法，这些资源包括 Base64 编码、MD5/SHA1 加密、UUID 以及队列（Queue）管理等。

FreeSWITCH 为了防止潜在的命名空间冲突等因素，对所有使用到的 APR 函数又进行了一些封装，这样所有的核心函数就都有了一致的命名空间“switch_”。这些封装在 switch_apr.c 里实现。

20.3.2 SWITCH APR

FreeSWITCH 采用了 APR 的代码风格及约定，非常易于使用。所以如果熟悉 APR 的话，看 FreeSWITCH 的源代码就容易多了。当然，反过来，熟悉了 FreeSWITCH 的源代码也会熟悉 APR。

1. 命名空间

在 APR 和 APR-UTIL 中，所有的公开接口都使用了字符串前缀“apr_”（数据类型和函数）和“APR_”（宏）。与此类似，在 FreeSWITCH 中，所有的核心接口函数也都使用了“switch_”前缀的函数和“SWITCH_”前缀的宏。

[1] 参见 http://apr.apache.org/。

[2] 参见 http://httpd.apache.org。

在 APR 命名空间中，也大量使用了二级命名空间，如“apr_socket_”等。同理在 FreeSWITCH 中，也有类似的如“switch_file_”、“switch_core_session_”等二级及三级命名空间。

2. 声明的宏

APR 使用类似于 APR_DECLARE 的宏进行声明，例如：

```
APR_DECLARE(apr_status_t) apr_initialize(void);
```

在很多的平台上，这是一个空声明，并且扩展为：

```
apr_status_t apr_initialize(void);
```

但在某些平台上，如在 Windows 的 Visual C++ 平台上，需要使用特有的、非标准的关键字，例如“_dllexport”、“__stdcall”等来允许其他的模块使用一个函数，这些宏就需要扩展以适应这些需要的关键字。

与 APR 类似，在 FreeSWITCH 中，大部分使用 SWITCH_DECLARE 或 SWITCH_DECLARE_DATA 之类的声明。

另外，FreeSWITCH 中不同类型的模块也都有专用的声明，如声明 Application 的 SWITCH_STANDARD_APP、声明 Dialplan 的 SWITCH_STANDARD_DIALPLAN 以及声明 API 的 SWITCH_STANDARD_API 等。

大部分的宏以及常量、枚举等都在 switch_types.h 中定义。

3. apr_status_t 和返回值

在 APR 中广泛采用的一个约定是：函数返回一个状态值，用来为调用者指示成功或者返回一个错误代码。这个类型便是 apr_status_t，它是在 apr_errno.h 中定义的整数值。因此一个 APR 函数的常见原型就是：

```
APR_DECLARE(apr_status_t) apr_do_something(...function args...);
```

返回值应当在逻辑上进行判断，并进行相应的错误处理。返回值 APR_SUCCESS 意味着成功。FreeSWITCH 也定义了类似的返回值，并以 SWITCH_SUCCESS 对应 APR_SUCCESS。这里注意一点，SWITCH_SUCCESS 对应该的枚举值为 0，因此常见的错误是：

```
apr_status_t status;

status = call_some_function(... args ...);
if (status) {/* 成功? */
    return status;
} else {
    ...
}
```

上面的程序片断是错误的，如果要判断执行结果是否成功，应该永远使用下面的方式：

```
if (status == SWITCH_STATUS_SUCCESS)
```

在 FreeSWITCH 的历史上也曾经因为这个原因出过 Bug（高手也会犯错误）。

另外，有些函数会返回一个字符串（char * 或者 const char *）、一个 void * 或者 void，这些函数就被认为没有失败条件或者在发生错误时返回一个空指针。

4. 内存池

APR 使用内存池来方便对内存的管理。大家都知道，C 语言中的内存管理是“臭名昭著”的。而在 APR 中，通过使用内存池，用户在申请内存时可以不用时刻记着释放申请到的内存，可以在用完后一起释放，这极大地方便了内存管理，并能防止产生大量内存碎片。

实际上，APR 中大部分的函数及资源严重依赖于内存池，如创建一个 Socket 需要内存池，创建一个 Thread 也需要内存池。这种情况听起来似乎有些过分，甚至其作者都认为这些操作显式地依赖于内存池是个巨大错误[⊖]，并希望能在 2.0 时解决这个问题。

内存池一般设计用于小的内存分配，如果要申请几兆字节的内存，那么不建议在内存池中申请[⊜]。

除了这些之外，APR 在使用起来还是相当方便的。在 APR 中通常认为从内存池中申请的内存分配永远不会失败。这个假设成立的原因在于如果内存分配失败，那么系统是不可恢复的，任何错误处理都将失败。

5. 其他

另外，SWITCH APR 还包装了 APR 中的字符串处理、文件管理、队列、互斥锁、Socket、线程库等。

20.3.3 main 函数

在此，回答一下本章开头提出的问题——关于看源代码从哪里开始的问题，答案就是从 main 函数开始看。

大家都知道，C 语言程序的执行都是从 main 开始执行的，FreeSWITCH 也不例外。打开 src/switch.c，在第 482 行就可以看到 main 函数。它的主要作用就是解析从命令行带来的各种参数，然后把一些重要参数记到一个 switch_core_flag_t 结构体中。默认系统会启动在前台。在 Linux 平台上，如果执行的时候提供了“-nc”参数（第 750 行），则在 UNIX 类系统上会通过 fork 系统调用将服务启动到后台（第 1701 行将最终调用 fork）；在 Windows 平台上，则是通过 FreeConsole WINAPI 实现的。

⊖ 原文如下：因为有人已经这样指出了，我承认 APR 完全依赖于内存池绝对是我在做 APR 时犯下的最大的错误。上个月我为 Linux Magazine 写了一篇文章，对这个错误说得很清楚。我个人对 APR 2.0 的目标是完全将内存池与 APR 分离，如果你想使用内存池的话，也非常简单，但使用内存池将不再是强制的。并且，内存池的使用也将是跟申请它的代码或函数相关的，而不是整个应用程序都使用同样的设置。参见：http://mail-archives.apache.org/mod_mbox/apr-dev/200502.mbox/%3C1f1d9820502241330123f955f@mail.gmail.com%3E。——笔者译

⊜ APR 文档中说：注意，使用 apr_palloc() 申请内存是并没有大小限制的。但是，使用它来申请大段的内存是不明智的。这是由于内存池是专门针对小内存设计的。实际上，一个内存池的最初大小是 8KB。如果你需要大段内存的话（如几兆字节），那么不应该使用内存池。参见：http://dev.ariel-networks.com/apr/apr-tutorial/html/apr-tutorial-3.html。——笔者译。

总之，不管是在前台还是后台，它在初始化一些环境并设置好系统相关的路径后，就执行第 1165 行，调用 switch_core_init_and_modload 函数加载各种模块。具体加载哪个模块则依赖于安装目录中的配置文件 conf/autoload_configs/modules.conf 中的设置。

```
482   int main(int argc, char *argv[])
483   {
...
750           else if (!strcmp(local_argv[x], "-nc")) {
751               nc = SWITCH_TRUE;
752           }
...
1066      if (nc) {
1067  #ifdef WIN32
1068          FreeConsole();
1069  #else
1070          if (!nf) {
1071              daemonize(do_wait ? fds : NULL);
1072          }
1073  #endif
...
1165      if (switch_core_init_and_modload(flags, nc ? SWITCH_FALSE : SWITCH_
            TRUE, &err) != SWITCH_STATUS_SUCCESS) {
```

加载完所有模块后，系统核心进入 switch_core.c:989 的 switch_core_runtime_loop，对于后台启动的实例来讲，它基本什么都不做，在 Windows 平台上，执行第 1001 行的 WaitForSingleObject 以等待服务终止；在 UNIX 类平台上，就是无限循环（第 1005 ~ 1007 行），其中第 1006 行相当于 sleep 1 秒；对于从前台启动的系统，它会在第 1011 行进入 switch_console_loop 以启动一个控制台接收用户的链盘输入并打印系统运行的信息（命令输出和日志等）。

```
989   SWITCH_DECLARE(void) switch_core_runtime_loop(int bg)
990   {
...
1001              WaitForSingleObject(shutdown_event, INFINITE);
...
1005          while (runtime.running) {
1006              switch_yield(1000000);
1007          }
...
1011          switch_console_loop();
```

switch_console_loop 函数在 switch_console.c:1098 定义。它使用跨平台的 libedit 库用于接收用户的按键并在控制台上打印信息。在第 1176 行，启动了一个新线程执行 console_thread 函数。

```
1098  SWITCH_DECLARE(void) switch_console_loop(void)
1099  {
...
1176      switch_thread_create(&thread, thd_attr, console_thread, pool, pool)
```

在 console_thread 中（第 1044 行），也是一个循环用于接收用户输入。如果用户输入一

条命令，则在检查命令的合法性后将命令放入命令历史记录（第 1075 行），以备以后再执行时可以使用键盘上的箭头键翻查命令历史。然后，在第 1076 行调用 switch_console_process 执行输入的命令并返回结果。

```
1044  static void *SWITCH_THREAD_FUNC console_thread(switch_thread_t *thread,
        void *obj)
1045  {
...
1075                  history(myhistory, &ev, H_ENTER, line);
1076                  running = switch_console_process(cmd);
```

switch_console_process（第 134 行）又调用了 switch_console_execute（第 348 行），后者最终在第 392 行调用核心提供的 switch_api_execute（switch_loadable_module.c:2282）执行输入的命令。

```
392          status = switch_api_execute(cmd, arg, NULL, istream);
```

如果用户在命令行上输入 sofia status，则上述命令执行的结果就是：

```
status = switch_api_execute("sofia", "status", NULL, istream);
```

上述命令的执行结果将存放到 istream 中，最终会在某处被取出并打印到命令行上。

20.3.4 可加载模块

通过对上一节的学习，我们的 FreeSWITCH 已经启动并可以接收和执行命令了。不过，我们还是往回倒一下，看看 FreeSWITCH 中的可加载模块是如何被加载的。

FreeSWITCH 的核心代码非常紧凑，大部分实际的功能都是由外围的模块实现和扩展的。

我们在上一节提到 switch_core_init_and_modload 函数负责初始化和加载各种模块。它是在 switch_core.c:2084 中定义的。在该文件的 2110 行，完成一些初始化后它就调用 switch_loadable_module_init 进行模块的初始化，该函数是在 switch_loadable_module.c:1747 中定义的。在第 1761 行到 1770 行定义了各种不同平台上的动态库的扩展名，如在 Windows 上动态链接库的扩展名是 .dll、在 Mac 上是 .dylib、在其他各种 UNIX 类系统上是 .so。

```
1747  SWITCH_DECLARE(switch_status_t) switch_loadable_module_init(switch_bool_
        t autoload)
1748  {
...
1761  #ifdef WIN32
1762      const char *ext = ".dll";
1763      const char *EXT = ".DLL";
1764  #elif defined (MACOSX) || defined (DARWIN)
1765      const char *ext = ".dylib";
1766      const char *EXT = ".DYLIB";
1767  #else
1768      const char *ext = ".so";
1769      const char *EXT = ".SO";
1770  #endif
```

在第 1772 行，初始化了一个结构体变量 loadable_modules，它是一个可加载模块的容器。

```
1772        memset(&loadable_modules, 0, sizeof(loadable_modules));
```

loadable_modules 的定义如下（switch_loadable_module.c:59）：

```
0059 struct switch_loadable_module_container {
0060     switch_hash_t *module_hash;
0061     switch_hash_t *endpoint_hash;
0062     switch_hash_t *codec_hash;
0063     switch_hash_t *dialplan_hash;
0064     switch_hash_t *timer_hash;
0065     switch_hash_t *application_hash;
0066     switch_hash_t *chat_application_hash;
0067     switch_hash_t *api_hash;
0068     switch_hash_t *json_api_hash;
0069     switch_hash_t *file_hash;
0070     switch_hash_t *speech_hash;
0071     switch_hash_t *asr_hash;
0072     switch_hash_t *directory_hash;
0073     switch_hash_t *chat_hash;
0074     switch_hash_t *say_hash;
0075     switch_hash_t *management_hash;
0076     switch_hash_t *limit_hash;
0077     switch_mutex_t *mutex;
0078     switch_hash_t *secondary_recover_hash;
0079     switch_memory_pool_t *pool;
0080 };
```

可以看出，它主要定义了各种哈希表（hash）。将来，新加载的各种模块将统一由不同的哈希表管理，如 mod_sofia 将被记入 endpoint_hash，mod_g729 将被记入 codec_hash 等。

此外，它还使用互斥（mutex）来防止多线程访问。pool 是一个内存池。在接下来的 1773 行就紧接着初始化了这个内存池：

```
1773        switch_core_new_memory_pool(&loadable_modules.pool);
```

第 1780 ~ 1798 行则初始化了所有的 hash 及 mutex。其中有些 hash 的键（Key）是区分大小写的，用 switch_core_hash_init 初始化；有些则是大小写无关的，用 switch_core_hash_init_nocase 进行初始化。这些数据结构初始化时都需要一个内存池，由此可以看出内存池的重要性㊀。

```
1780        switch_core_hash_init(&loadable_modules.module_hash, loadable_modules.pool);
1781        switch_core_hash_init_nocase(&loadable_modules.endpoint_hash,
              loadable_modules.pool);
...
1798        switch_mutex_init(&loadable_modules.mutex, SWITCH_MUTEX_NESTED,
              loadable_modules.pool);
```

系统加载完以后，就开始加载各种模块了。在 1802 ~ 1803 行，首先加载的是 CORE_SOFTTIMER_MODULE 和 CORE_PCM_MODULE 两个模块，这两个模块是直接在核心代

㊀ 如果继续跟踪这些函数，就会看到大部分最终会调用 APR 版本的函数，如 switch_mutex_init() 将最终调用 apr_thread_mutex_create()（switch_apr.c:2933）。唯一例外的是 hash，它使用了 SQLite3 提供的 hash 库。

码中实现的，因而比较特殊。

```
1802        switch_loadable_module_load_module("", "CORE_SOFTTIMER_MODULE",
SWITCH_FALSE, &err);
1803        switch_loadable_module_load_module("", "CORE_PCM_MODULE", SWITCH_
FALSE, &err);
```

我们暂且不深入研究这两个模块是如何加载的，继续往下走。第 1806 行，switch_xml_open_cfg() 将打开 XML 配置文件中的 modules.conf（参见 1753 行）部分（默认在安装目录 conf/autoload_configs/modules.conf.xml 中配置），经过一个 for 循环（1809 行）依次取得需要加载的模块的名字，并最终在第 1824 行执行 switch_loadable_module_load_module_ex 加载它们。

```
1806      if ((xml = switch_xml_open_cfg(cf, &cfg, NULL))) {
...
1809              for (ld = switch_xml_child(mods, "load"); ld; ld = ld->next) {
...
1824                  if (switch_loadable_module_load_module_ex((char *) path,
(char *) val, SWITCH_FALSE, global, &err) == SWITCH_STATUS_GENERR) {
```

然后，用同样的方法尝试加载 post_load_modules.conf（参见第 754 行）中配置的模块（1839 行起）。

另外，如果上面两个配置文件中都没有找到可加载的模块（1867），则尝试加载所有模块 (1872~1897 行)。

```
1867      if (!count) {
1869          all = 1;
1870      }
1871
1872      if (all) {
...
1897                  switch_loadable_module_load_module((char *) SWITCH_GLOBAL_
dirs.mod_dir, (char *) fname, SWITCH_FALSE, &err);
```

无论如何，模块加载完毕后，将执行 switch_loadable_module_runtime 函数（第 1902 行）。该函数在第 114 行定义，关于该函数的作用我们在下一节讲解。

```
1902      switch_loadable_module_runtime();
```

接下来，我们看一下模块是怎么被加载的。模块加载最终是由 1476 行的 switch_loadable_module_load_module_ex 实现的。它会首先计算与欲加载的模块对应的文件名，并在第 1515 行检查 loadable_modules.module_hash 这个哈希表来判断该模块是否已被加载，如果未被加载，则在第 1519 行调用 switch_loadable_module_load_file 将该模块加载到内存。

```
1476  static switch_status_t switch_loadable_module_load_module_ex(
        char *dir, char *fname, switch_bool_t runtime,
        switch_bool_t global, const char **err)
1477  {
...
1515      if (switch_core_hash_find_locked(loadable_modules.module_hash,
```

```
                file, loadable_modules.mutex)) {
...
1519        } else if ((status = switch_loadable_module_load_file(
              path, file, global, &new_module)) == SWITCH_STATUS_SUCCESS) {
```

switch_loadable_module_load_file 是在第 1328 行定义的。这个函数我们需要好好看一下。

在第 1356 行和第 1360 行，它会多次尝试使用 switch_dso_open 来打开相应的模块的动态链接库（switch_dso_open 在 switch_dso.c:35 中定义。在 Windows 平台上，它将使用 LoadLibraryEx 来打开相应的 .dll 库，在 Mac 和 Linux 上，它将使用 dlopen 函数打开相应的 .so 文件）。

```
1328  static switch_status_t switch_loadable_module_load_file(char *path,
        char *filename, switch_bool_t global, switch_loadable_module_t **new_module)
1329  {
...
1356          dso = switch_dso_open(lib_path, load_global, &derr);
...
1360      dso = switch_dso_open(NULL, load_global, &derr);
```

动态库打开后，通过下面的代码找到动态库里的符号表（第 1379 行）。执行到 1402 行时，如果符号表加载成功，则将该符号表赋值给 mod_interface_functions 变量。

```
1379              interface_struct_handle = switch_dso_data_sym(dso, struct_name, &derr);
...
1402              mod_interface_functions = interface_struct_handle;
```

mod_interface_functions 是一个 switch_loadable_module_function_table 结构的结构体，它是在 switch_type.h:2228 中定义的：

```
2228  typedef struct switch_loadable_module_function_table {
2229      int switch_api_version;
2230      switch_module_load_t load;
2231      switch_module_shutdown_t shutdown;
2232      switch_module_runtime_t runtime;
2233      switch_module_flag_t flags;
2234  } switch_loadable_module_function_table_t;
2235
```

该结构体定义了几个指向函数的指针，分别是 load、shutdown 和 runtime。如果被加载的模块中实现了这些函数，则这些指针指向相关的函数入口，如果没有实现，就是 NULL。

每个模块都必须实现 load 函数，它一般用于模块的初始化操作。因而在第 1403 行，load 函数的指针被赋值给 load_func_ptr 这个变量。

```
1403              load_func_ptr = mod_interface_functions->load;
```

第 1411 行，load 函数将被执行：

```
1411          status = load_func_ptr(&module_interface, pool);
```

load 函数的原型是使用 SWITCH_MODULE_LOAD_FUNCTION(name) 这个宏来定义的（switch_types.h:2211），该宏展开的结果就是：

```
switch_status_t mod_xx_load(
```

```
switch_loadable_module_interface_t **module_interface, switch_memory_pool_t *pool)
```

因而，如果该函数执行成功，将返回 SWITCH_STATUS_SUCCESS，并且会初始化一个 module_interface 指针。然后，在第 1424 行初始化一个 module 变量。

```
1424        if ((module = switch_core_alloc(pool,
                  sizeof(switch_loadable_module_t))) == 0) {
```

module 变量是一个如下所示的结构（在第 44 行定义）。它定义了该模块的一些参数，其中，成员 module_interface 就用于存放刚刚在第 1411 行初始化的 module_interface 指针。

```
44  struct switch_loadable_module {
45      char *key;
46      char *filename;
47      int perm;
48      switch_loadable_module_interface_t *module_interface;
49      switch_dso_lib_t lib;
50      switch_module_load_t switch_module_load;
51      switch_module_runtime_t switch_module_runtime;
52      switch_module_shutdown_t switch_module_shutdown;
53      switch_memory_pool_t *pool;
54      switch_status_t status;
55      switch_thread_t *thread;
56      switch_bool_t shutting_down;
57  };
```

接下来从第 1451 行开始，对 module 中的各成员赋值。最后，在第 1463 行初始化 new_module 指针，返回到调用该函数的地方，模块加载成功。

```
1451      module->pool = pool;
1452      module->filename = switch_core_strdup(module->pool, path);
1453      module->module_interface = module_interface;
1454      module->switch_module_load = load_func_ptr;
...
1463      *new_module = module;
```

总之，模块加载的流程就是，首先找到模块对应的动态库文件，然后打开并找到符号表，接下来执行模块中的 load 函数。另外，如果模块定义了 runtime 及 shutdown 函数，也将一并记录到 module 结构的 switch_module_runtime 及 switch_module_shutdown 成员变量中。

switch_loadable_module_load_file 执行完毕后，得到了一个 new_module 指针，并返回到第 1519 行。紧接着在第 1520 行执行 switch_loadable_module_process 函数，它使用模块的文件名（file）和我们新得到的 new_module 结构作为参数传入。

```
1519      } else if ((status = switch_loadable_module_load_file(
            path, file, global, &new_module)) == SWITCH_STATUS_SUCCESS) {
1520        if ((status = switch_loadable_module_process(file, new_module))
```

switch_loadable_module_process 函数是在第 133 行定义的，在第 138 行，有如下语句：

```
138      new_module->key = switch_core_strdup(new_module->pool, key);
```

第 138 行初始化 new_module 的 key，它是一个字符串，实际上就是传入的文件名。switch_core_strdup 用于制作一个 key 的副本（duplicate），它需要的内存是从内存池中申请

的，因而后续不需要明确释放，在模块卸载时直接释放掉内存池就行了。

在第 140 行通过互斥的 mutex 来锁定全局的 loadable_modules 结构，并在第 141 行向其中的 loadable_modules.module_hash 哈希表中插入该模块，以记录该模块被加载了。

```
140     switch_mutex_lock(loadable_modules.mutex);
141     switch_core_hash_insert(loadable_modules.module_hash, key, new_module);
```

第 143 行进行判断，如果被加载的模块实现了一个 endpoint_interface，则在第 150 行将它记录到 loadable_modules.endpoint_hash 中。

```
143     if (new_module->module_interface->endpoint_interface) {
...
150                 switch_core_hash_insert(loadable_modules.endpoint_hash,
ptr->interface_name, (const void *) ptr);
```

第 151 行产生一个模块加载的事件——SWITCH_EVENT_MODULE_LOAD，并于第 156 行发送出去：

```
151                 if (switch_event_create(&event, SWITCH_EVENT_MODULE_LOAD)
== SWITCH_STATUS_SUCCESS) {
...
156                     switch_event_fire(&event);
```

同理，如果该模块也实现了其他的 interface（在同一模块中可以实现多个 interface，如第 163 行的 codec_interface，第 220 行的 dialplan_interface 等），则都记录到相应的哈希表中，产生相关的事件，并针对不同的 interface 类型可能还有不同的检查和其他处理等。

```
163     if (new_module->module_interface->codec_interface) {
...
220     if (new_module->module_interface->dialplan_interface) {
```

至此，模块就加载成功了，最后到第 551 行会将锁定的临界区的资源解锁，并返回成功的状态码。

```
551     switch_mutex_unlock(loadable_modules.mutex);
552     return SWITCH_STATUS_SUCCESS;
```

不过，模块加载成功并不表示所有工作已完成。上面的函数返回后又回到第 1521 行。接下来，判断如果新加载的模块定义了 runtime 函数，则启动一个新的线程，执行该 runtime 函数。

```
1521    if (new_module->switch_module_runtime) {
1522        new_module->thread = switch_core_launch_thread(
                switch_loadable_module_exec, new_module, new_module->pool);
1523    }
```

runtime 函数是循环的执行的，即只要 runtime 函数不返回 SWITCH_STATUS_TERM，则它就会被再次执行，直到该模块被卸载为止，参见第 99 ~ 101 行。

```
99      for (restarts = 0; status != SWITCH_STATUS_TERM && !module->shutting_
down; restarts++) {
100         status = module->switch_module_runtime();
101     }
```

至此，模块加载的工作才算完成了。FreeSWITCH 进入正常运行阶段。

20.3.5 模块的结构

上一节，我们讲了核心中对可加载模块的处理，本节我们接着来看一下可加载模块的结构。

在 src/mod/sdk/autotools/src 目录下有一个 mod_example.c，其描述了一个最精简的模块的结构。其他模块可以在这个基础上修改。该文件只是一个例子，有些语句默认是注释掉的，在需要的时候可以打开。

该文件不长，因此我们把它全部的内容都列在这里（除了最前面的版权信息及最后面对编译器的注释）。真正的代码是从第 33 行开始的（为了与实际文件对应，我们为空行也编了行号）。它首先装入 switch.h，使得它可以引用 FreeSWITCH 核心中的公用函数（即所谓的 Core Public API）。

然后，它使用 3 个宏（在 switch_types: 第 2211 ~ 2213 行定义）分别声明了 3 个函数：第 36 行的 mod_example_shutdown、第 37 行的 mod_example_runtime 以及第 40 行的 mod_example_load。其中，只有 load 函数是必需的，因此其他两个默认是注释掉的。

上面只是对 3 个函数的前向声明，真正让这 3 个函数起作用的行是第 41 行。它的作用是告诉核心，在加载该模块时，就要回调本模块的 mod_example_load 函数进行一些初始化操作（即我们上一节讲过的 switch_loadable_module.c:1411）。

```
33  #include <switch.h>
34
35  /*
36  SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_example_shutdown);
37  SWITCH_MODULE_RUNTIME_FUNCTION(mod_example_runtime);
38  */
39
40  SWITCH_MODULE_LOAD_FUNCTION(mod_example_load);
41  SWITCH_MODULE_DEFINITION(mod_example, mod_example_load, NULL, NULL);
42
```

在这里，我们仅使用了 load 回调函数，如果把其他两个都用上，我们可以这样写：

```
SWITCH_MODULE_DEFINITION(mod_example,
    mod_example_load, mod_example_shutdown, mod_example_runtime);
```

这样的话，当模块被加载的时候就会回调 load、启动一个新线程运行 runtime，并在模块被卸载的时候执行 shutdown 函数。

当该模块被加载时（如在 FreeSWITCH 控制台上执行 mod_example），则回调其下面的函数。该函数在参数中会传过来一个空指针（实际上一个个双重指针）——module_interface，我们需要初始化这个指针（第 46 行）。在 module_interface 初始化完成后打印一条日志（第 48 行），并返回 SWITCH_STATUS_SUCCESS 值（第 51 行）以表示初始化成功。如果初始化失败（如不能连接数据库、不能申请相关资源等，都可能导致初始化失败），则可以返回 SWITCH_STATUS_FALSE 或其他错误值。

```
43 SWITCH_MODULE_LOAD_FUNCTION(mod_example_load)
44 {
45     /* connect my internal structure to the blank pointer passed to me */
46     *module_interface = switch_loadable_module_create_module_interface(pool,
       modname);
47
48     switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_NOTICE, "Hello World!\n");
49
50     /* indicate that the module should continue to be loaded */
51     return SWITCH_STATUS_SUCCESS;
52 }
53
```

下面是模块卸载时的回调函数，其可以用于断开数据库连接、释放内存、清理相关现场等。在此，我们的模块没有申请什么资源，因而直接返回成功（第 58 行）。

```
54 /*
55   Called when the system shuts down
56 SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_example_shutdown);
57 {
58     return SWITCH_STATUS_SUCCESS;
59 }
60 */
61
```

如果 runtime 函数存在，则系统核心会启动一个新线程来调用该函数。在这里，可以是一个无限循环（如第 67 行，当然要记住给无限循环终止条件，否则该模块就不能卸载了），也可以在执行一段时间后返回一个状态值，只要返回值不是 SWITCH_STATUS_TERM，该函数就会被再次调用。

```
62 /*
63   If it exists, this is called in it's own thread when the module-load completes
64   If it returns anything but SWITCH_STATUS_TERM it will be called again automaticly
65 SWITCH_MODULE_RUNTIME_FUNCTION(mod_example_runtime);
66 {
67     while(looping)
68     {
69         switch_yield(1000);
70     }
71     return SWITCH_STATUS_TERM;
72 }
73 */
```

上述就是在 FreeSWITCH 中可加载模块的大体结构。我们在后面会看到编写一个新模块的实际例子。

20.3.6 Session 和 Channel

在 FreeSWITCH 核心中，与通话最相关的部分莫过于 Session 和 Channel 了。FreeSWITCH 是一个 B2BUA，因此，它参与通话的每一条腿（Leg）都是一个 Channel。而 Session 则比 Channel 更高级一些，它用于描述一次会话，也就是说，虽然 Session 与 Channel 总是一一对应的，但前者管的事更多一些。也可以这样认为，Session 更关注于控制信令层，而

Channel 更关注于媒体层。

每当有一个电话到来时，或者每次从 FreeSWITCH 中发起一路通话时，便建立一个 Session（同时生成一个 Channel）。

用于标志 Session 的是一个 struct switch_core_session 的结构体，它的部分定义如下：

```
107 struct switch_core_session {
108     switch_memory_pool_t *pool;
109     switch_thread_t *thread;
110     switch_thread_id_t thread_id;
111     switch_endpoint_interface_t *endpoint_interface;
112     switch_size_t id;
113     switch_session_flag_t flags;
114     switch_channel_t *channel;
...
```

可以看出，在 switch_core_session 中有一个指向 channel 的指针（第 114 行）。上述的定义在是 src/include/private/switch_core_pvt.h 中实现的。之所以在 private（私有的）下面定义，是因为它不想让 FreeSWITCH 核心之外的应用知道 Session 中的细节。也就是说，其他系统如果使用 FreeSWITCH 的库，即使在 FreeSWITCH 内部的模块中，也看不到 Session 内部的东西。如果一段代码需要知道与 Session 相关的 Channel，则只能用 switch_core_session_get_channel(session) 函数从 session 变量中取得，而不能直接调用 session->channel。如果这样说读者还不是太明白的话，看一看该文件前面的注释：

```
 * switch_core.h -- Core Library Private Data (not to be installed into the system)
 * If the last line didn't make sense, stop reading this file, go away!,
 * this file does not exist!!!!
```

它的大意是说，该文件的内容是私有的，不需要看，就当它不存在！因而，我们也没必须深入研究了。

Session 是由 switch_core_session_request_uuid（第 2259 行）函数生成的。

```
2259    SWITCH_DECLARE(switch_core_session_t *) switch_core_session_request_uuid(
2260                    switch_endpoint_interface_t *endpoint_interface,
2261                    switch_call_direction_t direction,
2262                    switch_originate_flag_t originate_flags,
2263                    switch_memory_pool_t **pool, const char *use_uuid)
```

在该函数中，它会检查是使用现有的内存池（第 2319 行）还是创建一个新的内存池（第 2322 行）。然后在该内存池上为 session 结构体变量申请内存空间（第 2325 行），并将它的 pool 成员变量指向该内存池（第 2326 行）。这样我们就有了一个 Session 了，并且，以后与该 Session 有关的内存申请都可以在该内存池中申请。该内存池会在 Session 消亡时释放，因而在很大程度上方便了内存管理。

```
2318    if (pool && *pool) {
2319        usepool = *pool;
2320        *pool = NULL;
2321    } else {
2322        switch_core_new_memory_pool(&usepool);
```

```
2323        }
2324
2325        session = switch_core_alloc(usepool, sizeof(*session));
2326        session->pool = usepool;
2327
```

接下来我们看到，第 2330 行又在该内存池中又立即为与 session 所对应的 Channnel 申请了内存，并在第 2334 行对 Channel 进行了初始化，同时将 Channel 的当前状态设为 CS_NEW。

```
2330        if (switch_channel_alloc(&session->channel, direction, session->pool)
!= SWITCH_STATUS_SUCCESS) {
2331            abort();
2332        }
2333
2334        switch_channel_init(session->channel, session, CS_NEW, 0);
```

如果在调用该函数时提供了一个 UUID，则使用它（第 2344 行），否则自动生成一个（第 2346 行），它用于标志一个 Channel。接下来，就可以设置通道变量了，如第 2350 ~ 2351 行。

```
2343        if (use_uuid) {
2344            switch_set_string(session->uuid_str, use_uuid);
2345        } else {
2346            switch_uuid_get(&uuid);
2347            switch_uuid_format(session->uuid_str, &uuid);
2348        }
2349
2350        switch_channel_set_variable(session->channel, "uuid", session->uuid_str);
2351        switch_channel_set_variable(session->channel, "call_uuid", session->uuid_str);
```

接下来，进行初始化与 Session 相关的各种变量、申请相关内存、初始化 Mutex、锁、队列等操作。最后，将与该 Session 对应的 UUID 记录到一个核心的哈希表中（第 2381 行），至此，Session 的初始化基本上就结束了。

```
2381        switch_core_hash_insert(session_manager.session_table, session-
               >uuid_str, session);
2382        session->id = session_manager.session_id++;
2383        session_manager.session_count++;
```

然后，第 1876 行的 switch_core_session_thread_launch 函数会被调用，来启动一个新的线程。

```
1876 SWITCH_DECLARE(switch_status_t) switch_core_session_thread_launch(
                                        switch_core_session_t *session)
```

由于启动一个新的线程是比较费时的操作，因而在系统内部维护了一个线程池。在该函数第 1888 行，判断核心参数是否启用线程池（默认启用），如果是，则就在第 1889 行把该 Session 推到线程池队列中去。否则就在第 1906 行启动一个新线程，执行 switch_core_session_thread 函数（当然在线程池队列中找到一个可用的线程后，也会执行该函数）。

```
1888        if (switch_test_flag((&runtime), SCF_SESSION_THREAD_POOL)) {
```

```
1889          return switch_core_session_thread_pool_launch(session);
1890      }
...
1906          if (switch_thread_create(&thread, thd_attr, switch_core_session_thread,
            session, session->pool) == SWITCH_STATUS_SUCCESS) {
1907            switch_set_flag(session, SSF_THREAD_STARTED);
1908            status = SWITCH_STATUS_SUCCESS;
```

switch_core_session_thread是在第1555行定义的。它有两个传入参数，一个是当前线程的指针thread，另一个是一个无类型的（void *）指针obj，该obj实际上就是我们的Session指针，因此在第1557行，初始化了一个session变量并指向与obj指针同样的地址。在进行一些初始化操作后，便执行1565行的switch_core_session_run函数。

```
1555  static void *SWITCH_THREAD_FUNC switch_core_session_thread(switch_
thread_t *thread, void *obj)
1556  {
1557      switch_core_session_t *session = obj;
...
1565      switch_core_session_run(session);
```

转了这么一大圈，我们终于找到了关键的地方。switch_core_session_run实际上是一个状态机。该状态机的定义在switch_types.h中，它是一个枚举类型的定义，内容如下：

```
1179  typedef enum {
1180      CS_NEW,                 // 新建
1181      CS_INIT,                // 已初始化
1182      CS_ROUTING,             // 路由
1183      CS_SOFT_EXECUTE,        // 准备好执行，可由第三方控制
1184      CS_EXECUTE,             // 执行Dialplan中的App
1185      CS_EXCHANGE_MEDIA,      // 与另一个Channel在交换媒体
1186      CS_PARK,                // Park，等待进一步的命令指示
1187      CS_CONSUME_MEDIA,       // 消费掉媒体并丢弃
1188      CS_HIBERNATE,           // 没事可干，Sleep
1189      CS_RESET,               // 重置
1190      CS_HANGUP,              // 挂机，结束信令和媒体交互
1191      CS_REPORTING,           // 收集呼叫信息（如写CDR等）
1192      CS_DESTROY,             // 待销毁，退出状态机
1193      CS_NONE                 // 无效
1194  } switch_channel_state_t;
```

switch_core_session_run函数是在switch_core_state_machine.c:414中定义的。

```
414  SWITCH_DECLARE(void) switch_core_session_run(switch_core_session_t *session)
```

该函数主要的功能就是执行一个循环，只要该Session所对应的Channel的状态不是CS_DESTROY，它就会一直循环。

```
449      while ((state = switch_channel_get_state(session->channel)) != CS_
DESTROY) {
```

在循环体内使用了一些switch…case语句，用于决定在不同的状态执行哪些代码段或函数，部分代码如下。其中，有一些关键的代码段被提取出来，放到一个名为STATE_MACRO宏中执行了，而该宏就是状态机中最关键的部分。

```
          switch (state) {
          case CS_NEW:
...
          case CS_DESTROY:
              goto done;
          case CS_REPORTING:  /* Call Detail */
...
          case CS_HANGUP: /* Deactivate and end the thread */
...
          case CS_INIT:    /* Basic setup tasks */
              {
...
                  STATE_MACRO(init, "INIT");
...
          case CS_ROUTING:/* Look for a dialplan and find something to do */
              STATE_MACRO(routing, "ROUTING");
              break;
          case CS_RESET:      /* Reset */
              STATE_MACRO(reset, "RESET");
              break;
```

STATE_MACRO 宏是在第 356 行定义的，该宏比较复杂，其实展开以后，大致就想当于下面的代码片断（这里用伪代码表示）：

```
1 do {
2     switch_log_printf("INIT");
3     if (driver_state_handler->on_init) {
4             driver_state_handler->on_init(session);
5         }
6     }
7 } while (0)
```

这里以 CS_INIT 状态为例，从伪代码可以看出，该宏整个套在一个大的“do { ... } while (0)”单次循环体中⊖。如果在与该 Channel 对应的 Endpoint 的底层驱动中在当前状态上安装了回调函数，则会执行该回调函数（第 4 行）。

当然，实际的情况比这里的情况要复杂得多，它更类似于下面的样子。即，除了可以在 Channel 对应的 Endpoint 的底层驱动中安装回调函数外，在其他的模块或第三方应用中也可以在 Channel 上安装回调函数，以后在 Channel 的状态机状态发生变化时得到回调，那么下面这段伪代码就有机会执行额外的回调函数（第 5 行）了。

```
1 do {
2     switch_log_printf("INIT");
3     if (!driver_state_handler->on_init ||
        (driver_state_handler->on_init(session) == SWITCH_STATUS_SUCCESS ) {
4             while (do_extra_handlers) {
5                 do_extra_handlers(...);
6             }
7         }
8     }
9 } while (0)
```

⊖ 这是一个处理宏的技巧，有如支持定义局部变量、支持复杂的宏定义而不用担心产生副作用等多种好处，在 Linux 内核中就使用了这种技术，参见：http://kernelnewbies.org/FAQ/DoWhile0。

实际的情况其实比这个还要复杂，但这里我们就不深入研究了。总之，你只需要知道在 Channel 的核心状态机上可以安装回调函数，并在状态发生变化时得到回调。如果对细节特别感兴趣的读者也可以使用“gcc -E”命令将该源文件中的宏展开看一看。

从这一段的代码我们知道，Session 与 Channel 是息息相关的。初始化了 Session 之后，就有了 Channel，而状态机全部都是在 Channel 上实现的（其中 CS_INIT 中的 CS 便是 Channel State 的意思）。当然，核心中也定义了很多专门对 Channel 操作的函数，大部分都是在 switch_channel.c 中实现的。这些函数的名称和代码看起来都很直观，在这里我们就不多讲了，等到后面用到的时候再个别进行说明。

20.3.7 SWITCH IVR

大部分媒体处理逻辑都是在 switch_ivr_*.c 中实现的，其中多个源代码实现了不同的 switch_ivr 逻辑，如 switch_ivr_async.c 进行异步处理、switch_ivr_bridge.c 处理话路桥接等。在此，我们先来看一个简单的 echo 应用。关于 echo App 我想大家都已经很熟悉了，我们知道它的作用就是将收到的媒体（音频或视频）原样再发回去。下面我们就看一看它是怎么实现的。

在通话执行到 echo App 时，将最终执行到 switch_ivr_async.c:629 定义的 switch_ivr_session_echo 函数。由于 echo 应用是需要媒体的，如果在执行 echo 时电话还没有应答（如在 SIP 应用中还没有收到或发送“200 OK”），则它会在第 636 行调用 switch_channel_pre_answer 试图在电话应答之前建立媒体连接（如果在 SIP 应用中将发送带 SDP 的 183 消息以尝试建立媒体连接）。当然，这是一个小的细节，我们继续往下看。

```
629 SWITCH_DECLARE(switch_status_t) switch_ivr_session_echo(switch_core_
        session_t *session, switch_input_args_t *args)
630 {
...
636     if (switch_channel_pre_answer(channel) != SWITCH_STATUS_SUCCESS) {
637         return SWITCH_STATUS_FALSE;
638     }
```

在该函数中，第 644 行执行一个 while 循环，只要该 Channel 是正常的（由 switch_channel_ready 判断，它会检查一系列参数，在 Channel 正常建立时将返回真，在挂机或出现其他错误的情况下返回假），便会一直循环。然后，在第 645 行，调用核心的函数 switch_core_session_read_frame 从该 Channel 中读取一帧的数据（这里的一帧，在 SIP 应用中就是一个 RTP 包中的数据，如可能是 20 毫秒的音频数据）。接下来在第 646 行通过一个宏来判断读到的数据是否有效，如果无效就跳出循环（第 647 行）。如果数据有效，就继续进行。第 656 行用于处理该 Channel 上相关的事件，如检查 DTMF 等。在收到 DTMF 的情况下会调用相关的回调函数（第 665 行）。

```
644     while (switch_channel_ready(channel)) {
645         status = switch_core_session_read_frame(session, &read_frame,
SWITCH_IO_FLAG_NONE, 0);
646         if (!SWITCH_READ_ACCEPTABLE(status)) {
```

```
647                 break;
648             }
...
656             switch_ivr_parse_all_events(session);
...
665                 if (switch_channel_has_dtmf(channel)) {
...
```

我们跳过一些细节，直接看第 694 行，可以看到它又调用了 switch_core_session_write_frame，即将收到的数据写回了 Session 中，然后这些音频数据就会发到远端。当然，如果该 Channel 上有视频的话，它也会进行相关的处理，我们暂时忽略视频的处理代码。

总之，该函数主要的功能就是调用 switch_core_session_read_frame 读取音频数据，并通过 switch_core_session_write_frame 写回去。有关这两个函数的实现我们将在下一节讲到。

20.3.8 Core IO

在上一节我们讲到，switch_core_session_read_frame 用于读数据，而 switch_core_session_write_frame 用于写数据。这两个函数是在 switch_core_io.c 中定义的。它们都非常长，因此我们在此只简单分析一下关键点。

switch_core_session_read_frame 是在第 147 行定义的。

```
147  SWITCH_DECLARE(switch_status_t) switch_core_session_read_frame(
                    switch_core_session_t *session, switch_frame_t **frame,
                    switch_io_flag_t flags, int stream_id)
```

一般来说，在 SIP 应用中，每 20 毫秒就会读到一帧音频数据（但也不是绝对的）。如果在读的过程中读不到数据但又不至于产生错误，该函数就返回一个静音包（Comfort Noise Generation，CNG，可以根据它产生舒适噪音）。

```
167             *frame = &runtime.dummy_cng_frame;
168             return SWITCH_STATUS_SUCCESS;
```

除此之外，该函数中还有很多检查判断，我们就不详细看了。下面直接跳到第 246 行。该行会调用底层的 Endpoint 提供的 read_frame 回调函数来读数据。由于 FreeSWITCH 支持不同的 Endpoint，因此，这里使用回调函数的机制屏蔽各 Endpoint 的不同特性。比如，在 SIP 应用中，媒体数据将从 RTP 中读取，在 mod_portaudio 中数据是从本地声卡读取的，而在 mod_freetdm 应用中数据是从硬件的 TDM 板卡驱动中读取的。总之，核心层并不知道这些数据是从哪里来的，具体这一帧数据怎么来，得看底层的驱却是怎么实现的。

总之，第 246 行判断 Endpoint 底层的驱动是否实现了 read_frame 回调，如果实现了，就在第 249 行调用该回调函数读取数据。然后，判断当前的 Session 是否注册了其他的事件钩子（event_hook），如果注册了，也调用钩子里的 read_frame 回调函数（第 250 行）。也就是说，除了 Endpoint 之外，其他的函数或模块中也可以调用 switch_core_event_hook_add_* 一族的函数来安装相关的回调函数，以便在适当的时候得到回调。因此，在第 250 行，它便是检查是否有通过 switch_core_event_hook_add_read_frame 函数注册的钩子，如果有，就在第 251 行调用。

```
246      if (session->endpoint_interface->io_routines->read_frame) {
...
249          if ((status = session->endpoint_interface->io_routines->read_frame
                 (session, frame, flags, stream_id)) == SWITCH_STATUS_SUCCESS) {
250           for (ptr = session->event_hooks.read_frame; ptr; ptr = ptr->next) {
251               if ((status = ptr->read_frame(session, frame, flags,
                     stream_id)) != SWITCH_STATUS_SUCCESS) {
```

我们以前也提到过，录音等应用都是使用 Media Bug 来实现的，也就是往一个 Channel 上安装了一个 Media Bug，这就相当于给水管安装了一个“三通”。在第 309 行，就是检查该 Channel 上有没有安装“三通”，如果安装了，便在第 316 行一个一个把它们找出来，并调用它们指定的回调函数（第 341 行）。而在回调函数中可以取到我们在上面第 249 行读到的这一帧的数据，并使用它（在录音应用中，最典型的就是将这些声音数据写到录音文件中去）。

```
309      if (session->bugs && !((*frame)->flags & SFF_CNG) && !((*frame)->flags
             & SFF_NOT_AUDIO)) {
...
316          for (bp = session->bugs; bp; bp = bp->next) {
...
339                      if (bp->callback) {
340                          bp->native_read_frame = *frame;
341                          ok = bp->callback(bp, bp->user_data, SWITCH_ABC_
                                 TYPE_TAP_NATIVE_READ);
```

读者不要把这些回调搞混了。我们简单总结一下，第 249 行的 io_routines 中的回调是从底层的驱动中读媒体数据，而其他的如第 250 行的 event_hooks 中的回调及第 341 行中的 Media Bug 中的回调是使用这些数据。

接下来的代码中根据当前通话的各种不同的情况判断是否需要转码（由 need_codec 变量表示），如第 361 ~ 363 行就是其中一种判断条件：

```
361    if (session->read_codec->implementation->impl_id != codec_impl.impl_id) {
362             need_codec = TRUE;
363    }
```

如果需要转码，则程序执行到第 561 行就会执行解码操作。不管读到的数据是什么编码的（PCMU、PCMA、iLBC 等），都会先转换成一种中间的编码格式——L16（16 位的线性编码）。

```
464      if (switch_test_flag(session, SSF_READ_TRANSCODE) &&
             !need_codec && switch_core_codec_ready(session->read_codec)) {
...
561              status = switch_core_codec_decode(codec,
562                              session->read_codec,
563                              read_frame->data,
564                              read_frame->datalen,
565                              session->read_impl.actual_samples_per_second,
566                              session->raw_read_frame.data,
                                 &session->raw_read_frame.datalen,
                                 &session->raw_read_frame.rate,
567                              &read_frame->flags);
```

除了编解码转换外，FreeSWITCH 还支持码率的转换，如将 48000Hz 的高清音频转换成 8000Hz 的窄带音频等。第 792 行即是调用转码处理的函数。

```
789          if (session->read_resampler) {
792              switch_resample_process(session->read_resampler,
                     data, (int) read_frame->datalen / 2);
```

如果该 Session 当前的 read_codec 不是 L16，则这个时候就需要将 L16 线性编码的数据编码成 read_codec 指定的编码（即函数调用者实际需要的编码），该操作是在第 835 行调用的。

```
835              status = switch_core_codec_encode(session->read_codec,
836                                 enc_frame->codec,
837                                 enc_frame->data,
838                                 enc_frame->datalen,
839                                 session->read_impl.actual_samples_per_second,
840                                 session->enc_read_frame.data,
                                    &session->enc_read_frame.datalen,
                                    &session->enc_read_frame.rate, &flag);
```

switch_core_session_read_frame 这个函数非常长，里面还有更多的细节，我们就不一一研究了。总之，它的就要作用就是从底层的 Endpoint 驱动中读取一帧数据，然后调用各种回调函数，并将读到的数据返回（通过第 147 行的 frame 双重指针返回）。如果对方需要返回特定编码的数据，则在函数内部执行转码操作（解码和编码），以返回编码后的数据。

接下来我们简单看一下 switch_core_session_write_frame 函数，它是在 1025 行定义的：

```
1025  SWITCH_DECLARE(switch_status_t) switch_core_session_write_frame(
        switch_core_session_t *session, switch_frame_t *frame, switch_io_flag_t flags,
1026      int stream_id)
```

如果该 Session 处于 Proxy Packet 状态（SFF_PROXY_PACKET），则它会快速地在第 1072 行调用 perform_write 函数将数据发送出去：

```
1072        status = perform_write(session, frame, flag, stream_id);
```

否则的话，它也会进行一系列的检查，并进行必要的操作。它也会在必要的时候对数据进行解码（第 1173 行）和编码（第 1467 行）操作：

```
1173          status = switch_core_codec_decode(frame->codec, ...
...
1467          status = switch_core_codec_encode(session->write_codec, ...
```

或者进行码率转换：

```
1258              switch_resample_process(session->write_resampler, ...
```

或者对 Media Bug 回调进行处理：

```
1275      if (session->bugs) { ...
```

通过在写（write）的时候增加一个 Media Bug，可以在媒体数据实际发送出去之前对数据进行修改和替换（如果把读的 Media Bug 理解成录音和监听，那么写的 Media Bug 就可以理解为插话——在一个通话中突然插入另外一个人的语音或者播放一段音乐）。

在第 1415 行，switch_core_session_write_frame 也是调用 perform_write 将数据发送出去。

```
1415            status = perform_write(session, write_frame, flags, stream_id);
```

perform_write 函数是在第 961 行定义的，它最终将会调用 Endpoint 下 io_routines 中的 write_frame 回调函数并将数据发送出去（第 1013 行），然后在第 1015 行回调相关的 event_hooks 里的回调函数（如果有的话）。

```
1012        if (session->endpoint_interface->io_routines->write_frame) {
1013            if ((status = session->endpoint_interface->io_routines->write_frame(
                 session, frame, flags, stream_id)) == SWITCH_STATUS_SUCCESS) {
1014                for (ptr = session->event_hooks.write_frame; ptr; ptr = ptr->next) {
1015                    if ((status = ptr->write_frame(session, frame, flags, stream_id)) ...
```

上面我们讲了音频媒体的读写处理。如果是视频数据，则也有对应的 switch_core_media_read_video_frame 以及 switch_core_media_write_video_frame，实现逻辑都差不多。只是目前 FreeSWITCH 核心中还没有对视频转码的处理，因此代码要比音频部分简单得多。

总之，系统核心的 IO 操作屏蔽了底层的数据流读、写（收、发）细节，各种需要处理的媒体的应用只需要调用核心的 IO 函数进行数据的读、写操作，而不用考虑底层的不同。同时，这种架构使得增加一种新的 Endpoint 非常容易——只需要增加一个 Endpoint 的逻辑结构，安装相应的回调函数，并调用更底层的驱动程序或者协议库进行媒体流读、写即可。

20.3.9 Core Media

Core Media 用于在核心进行媒体协商和处理。这些代码原来是在 mod_sofia 模块中，但后来为了增加 WebRTC 的支持，把这一部分代码独立出来，放到了 switch_core_media.c 中。它目前主要是处理使用 SDP 描述的媒体（如基于 RTP 的媒体）。

如果 Endpoint 中需要 RTP 媒体支持，则它可以在 Session 中建立一个媒体句柄，然后通过 session->media_handle 来引用它。通过使用 Core Media，可以隐藏一个 SDP 媒体协商及 RTP 处理的细节，使得开发基于 RTP 的媒体程序更加简单。

在 Core Media 中，switch_core_media_read_frame 函数用于从底层的 RTP 中读一帧数据，其中，媒体的类型（type 参数）可以是 SWITCH_MEDIA_TYPE_AUDIO 或 SWITCH_MEDIA_TYPE_VIDEO（分别表示读音频数据和读视频数据）。在 Core Media 内部，有一个媒体引擎参数，它目前定义了音频和视频两个引擎组成的数组，在第 1414 行可以通过 engines[type] 找到所需要的媒体引擎。

```
1395    SWITCH_DECLARE(switch_status_t) switch_core_media_read_frame(
            switch_core_session_t *session, switch_frame_t **frame,
1396        switch_io_flag_t flags, int stream_id, switch_media_type_t type)
...
1414        engine = &smh->engines[type];
```

在第 1440 行，它调用 RTP 相关的函数 switch_rtp_zerocopy_read_frame 来读取一帧（也就是一个 RTP 包的数据）。

```
1440            status = switch_rtp_zerocopy_read_frame(engine->rtp_session,
                    &engine->read_frame, flags);
```

同样，switch_core_media_write_frame 函数（第 1767 行）则用于调用底层的 switch_rtp_write_frame 函数向对端发送 RTP 数据（第 1816 行）。

```
1767 SWITCH_DECLARE(switch_status_t) switch_core_media_write_frame(switch_
         core_session_t *session,
1768     switch_frame_t *frame, switch_io_flag_t flags, int stream_id, switch_
           media_type_t type)
...
1816      if (!switch_rtp_write_frame(engine->rtp_session, frame)) {
```

当然，除了媒体读写以外，Core Media 中还有 switch_core_media_negotiate_sdp 函数（用于媒体协商）、switch_core_media_activate_rtp 函数（用于启动 RTP 收发）等，我们在此就不多讲了。

20.3.10 Core RTP

FreeSWITCH 中的 RTP 媒体收、发都是在 switch_rtp.c 中实现的。我们在上一节提到过，在 Core Media 中，会调用 switch_rtp_zerocopy_read_frame 来读取一帧数据。所谓 zerocopy，就是不复制数据而直接返回一个数据指针。下面我们先来看一下这个函数。

switch_rtp_zerocopy_read_frame 函数是在第 5502 行定义的，它的输入参数 rtp_session 是一个 switch_rtp_t 类型的指针，它唯一标志了一个 RTP 连接。第二个参数 frame 是一个 switch_frame_t 类型的指针，它用于存放读到的数据。它主要是在第 5510 行调用 rtp_command_read 从底层的 Socket 中读取数据，并用读到的数据去填充 frame 指针指向的结构体。

```
5502 SWITCH_DECLARE(switch_status_t) switch_rtp_zerocopy_read_frame(
                  switch_rtp_t *rtp_session, switch_frame_t *frame,
                  switch_io_flag_t io_flags)
5503 {
...
5510      bytes = rtp_common_read(rtp_session, &frame->payload, &frame->pmap,
              &frame->flags, io_flags);
```

从第 5512 行开始，很容易看到 switch_frame_t 结构体的结构（该结构在 switch_frame.h:44 定义，我们就不再列出具体定义了）。其中，其成员变量 data 指向读到的数据；packet（第 5513 行）则指向 RTP 消息的开始（比 data 多一个 RTP 包头，一般是 12 个字节）；第 5514 行是 packet 的长度；第 5521 行是时间戳；第 5522 行是序号；第 5523 行是同步源标志；第 5524 行是 Marker 标志；第 5585 行是真正的媒体数据长度。

```
5512      frame->data = RTP_BODY(rtp_session);
5513      frame->packet = &rtp_session->recv_msg;
5514      frame->packetlen = bytes;
5515      frame->source = __FILE__;
5521      frame->timestamp = ntohl(rtp_session->recv_msg.header.ts);
5522      frame->seq = (uint16_t) ntohs((uint16_t) rtp_session->recv_msg.header.seq);
5523      frame->ssrc = ntohl(rtp_session->recv_msg.header.ssrc);
5524      frame->m = rtp_session->recv_msg.header.m ? SWITCH_TRUE : SWITCH_FALSE;
...
```

```
5585        frame->datalen = bytes;
5586        return SWITCH_STATUS_SUCCESS;
5587    }
```

rtp_common_read 是在第 4674 行定义的，它又根据不同的情况在第 4725 和第 4846 行分别调用 read_rtp_packet 来读取数据。

```
4674  static int rtp_common_read(switch_rtp_t *rtp_session,
                switch_payload_t *payload_type,
4675            payload_map_t **pmapP, switch_frame_flag_t *flags,
                switch_io_flag_t io_flags)
...
4725          status = read_rtp_packet(rtp_session, &bytes, flags, SWITCH_FALSE);
...
4846          status = read_rtp_packet(rtp_session, &bytes, flags, SWITCH_TRUE);
```

read_rtp_packet 又最终在第 4152 行调用 switch_socket_recvfrom 函数从真正的 Socket 中读取数据。switch_socket_recvfrom 仅是对 APR 库中的 apr_socket_recvfrom 的一个简单封装，后者会根据不同的平台调用不同的底层函数对 Socket 进行读取。

```
4137  static switch_status_t read_rtp_packet(switch_rtp_t *rtp_session,
                switch_size_t *bytes, switch_frame_flag_t *flags,
                switch_bool_t return_jb_packet)
4138  {
...
4152        status = switch_socket_recvfrom(rtp_session->from_addr,
                         rtp_session->sock_input, 0,
                         (void *) &rtp_session->recv_msg, bytes);
```

在上述的这一连串的读取函数中，有一连串的对读取到的数据进行检查的地方，保证了读取到的数据的正确性和安全性。同时，FreeSWITCH 中对有些不规范的 RTP 协议实现也适当进行了一些妥协，以便于跟那些设备对接⊖。

与读数据相反，在写数据时，则需要先初始化一个 switch_frame_t 的 frame 帧结构，然后调用 switch_rtp_write_frame 进行写（发送，第 6110 行）。当然，根据不同的条件，它或者在第 6514 行调用 switch_socket_sendto 直接发送，或者在第 6275 行调用 rtp_common_write 再进行深入的设置并发送，具体细节在此我们就不深入研究了。

```
6110  SWITCH_DECLARE(int) switch_rtp_write_frame(switch_rtp_t *rtp_session,
          switch_frame_t *frame)
6111  {
...
6154            if (switch_socket_sendto(rtp_session->sock_output,
                  rtp_session->remote_addr, 0, frame->packet, &bytes)
...
6275        return rtp_common_write(rtp_session, send_msg, data, len, payload,
                ts, &frame->flags);
```

前面我们说过，所有的 RTP 连接都是由一个 switch_rtp_t 类型的变量（如 rtp_session）

⊖ 甚至你可以在 switch_types.h 中发现一个 switch_rtp_bug_flag_t 枚举结构，里面列出了诸多已知的其他设备的 Bug，以及作者的抱怨。

来标志的。可以使用 switch_rtp_new 函数来新建一个 rtp_session，其函数定义如下：

```
SWITCH_DECLARE(switch_rtp_t *) switch_rtp_new(const char *rx_host,
        switch_port_t rx_port,
        const char *tx_host,
        switch_port_t tx_port,
        switch_payload_t payload,
        uint32_t samples_per_interval,
        uint32_t ms_per_packet,
        switch_rtp_flag_t flags[SWITCH_RTP_FLAG_INVALID],
        char *timer_name, const char **err, switch_memory_pool_t *pool)
```

其中，rx_host 及 rx_port 是本端的 IP 地址和端口号，tx_host 和 tx_port 则分别为远端的 IP 地址和端口号，至于其他参数的含义和用法可以参考源代码中具体使用该函数的地方（如 switch_core_media.c:4647），我们就不再赘述了。

当然，如果要释放一个 rtp_session，则可以使用下列函数：

```
SWITCH_DECLARE(void) switch_rtp_destroy(switch_rtp_t **rtp_session)
```

关于 RTP 的代码我们就介绍这么多，有兴趣的读者可以在这些基础上再深入研究。

20.3.11 SWITCH XML

通过前面的学习我们可以看到，FreeSWITCH 的配置文件严重依赖 XML。FreeSWITCH 对 XML 的解析是在 switch_xml 中实现的。

如果某个程序需要从 XML 中读取配置数据，则它会调用 switch_xml_open_cfg 函数首先来打开一个 XML 节点。该函数是在第 2392 行定义的。在该函数内部，它会在第 2400 行调用 switch_xml_locate 去查找相关的 XML 节点，并返回相关的 XML 结构指针。

```
2392  SWITCH_DECLARE(switch_xml_t) switch_xml_open_cfg(const char *file_path,
                                    switch_xml_t *node, switch_event_t *params)
2393  {
...
2400      if (switch_xml_locate("configuration", "configuration", "name", file_
              path, &xml, &cfg, params, SWITCH_FALSE) == SWITCH_STATUS_SUCCESS) {
2401          *node = cfg;
2402      }
```

switch_xml_locate 在第 1670 行定义。在该函数内部，它首先会判断一个 BINDINGS 全局变量中的链表结构。如果该链表非空，那么说明某个地方绑定了 XML 中的一个节点，能动态地提供 XML。回想一下我们 17.9 节讲到的 mod_xml_curl 的例子。那个例子在 mod_xml_curl 中，向核心的 XML 绑定了一个节点，然后每当执行到下面的函数需要一个这样的节点时，便回调 mod_xml_curl 中绑定的回调函数。这样的回调函数就是在第 1690 行执行的。

```
1670  SWITCH_DECLARE(switch_status_t) switch_xml_locate(const char *section,
1671      const char *tag_name,
1672      const char *key_name,
1673      const char *key_value,
1674      switch_xml_t *root, switch_xml_t *node, switch_event_t *params,
              switch_bool_t clone)
```

```
1675 {
...
1685     for (binding = BINDINGS; binding; binding = binding->next) {
...
1690         if ((xml = binding->function(section, tag_name, key_name, key_
                value, params, binding->user_data))) {
```

当然，如果没有动态绑定，或者获取动态绑定的 XML 资源时发生错误，则该函数会尝试从本地的 XML 配置文件中查找（第 1720 行）。

```
1718     for (;;) {
1719         if (!xml) {
1720             if (!(xml = switch_xml_root())) {
```

另外，关于详细的 XML 解析算法以及其他的细节我们在此就不多解释了。

20.3.12 SWITCH Event

FreeSWITCH 中有一些功能是事件驱动的。另外，事件也是 FreeSWITCH 内部与外部进行数据交换的载体。当 FreeSWITCH 中发生状态改变或者代码执行到某个阶段时，都会触发一些事件。同时，另外一些感兴趣的模块也可以订阅这些事件，以便在收到相应事件时执行相应的动作。从某种意义上说，这种事件机制与我们上面讲过的回调函数和钩子想要达到的效果是一样的，不同的是，事件采用“Pub/Sub”（即发布 / 订阅机制，又称生产者 / 消费者模型）建立的是一种更松的耦合关系，使用起来更方便、更自由。另外，外部的第三方系统也可以通过系统提供的接口订阅到事件，从而可以更容易地集成。

在系统初始化时，首先调用 switch_event_init 函数进行事件系统的初始化，该函数是在 switch_event.c:659 中定义的，它会初始化事件系统所需的内存池、哈希表、Mutex、队列等。

```
659 SWITCH_DECLARE(switch_status_t) switch_event_init(switch_memory_pool_t *pool)
```

在 event 初始化完成后，核心代码会调用 switch_event_launch_dispatch_threads（第 618 行定义）来启动事件分发的线程，这些线程最终会执行 switch_event_dispatch_thread 函数（第 645 行）。

```
618 SWITCH_DECLARE(void) switch_event_launch_dispatch_threads(uint32_t max)
619 {
...
645         switch_thread_create(&EVENT_DISPATCH_QUEUE_THREADS[index],
                thd_attr, switch_event_dispatch_thread, EVENT_DISPATCH_QUEUE, pool);
```

switch_event_dispatch_thread 函数定义在第 290 行。它内部执行一个无限循环，不断地从事件队列中取出一个个事件（第 322 行），然后在第 331 行调用 switch_event_deliver 分发出去。

```
290 static void *SWITCH_THREAD_FUNC switch_event_dispatch_thread(switch_
        thread_t *thread, void *obj)
291 {
...
314     for (;;) {
...
```

```
        if (switch_queue_pop(queue, &pop) != SWITCH_STATUS_SUCCESS) {
            continue;
        }

        event = (switch_event_t *) pop;
        switch_event_deliver(&event);
        switch_os_yield();
    }
```

switch_event_deliver 在第 391 行定义。它通过一个两层的 for 循环，不断判断所有的事件中有哪些事件被哪些节点（node）订阅了（第 398 ~ 400 行），如果有节点订阅，则调用订阅时提供的回调函数（第 402 行），即相当于把事件分发出去了。

```
391  SWITCH_DECLARE(void) switch_event_deliver(switch_event_t **event)
392  {
...
398          for (e = (*event)->event_id;; e = SWITCH_EVENT_ALL) {
399              for (node = EVENT_NODES[e]; node; node = node->next) {
400                  if (switch_events_match(*event, node)) {
401                      (*event)->bind_user_data = node->user_data;
402                      node->callback(*event);
```

当然，有节点订阅（消费）事件，就得有节点发布（生产）事件。在发布事件之前，首先要产生一个事件。产生事件是用 switch_event_create_subclass_detailed 实现的，为了使用方便，在 switch_event.h:381 中也定义了一个 switch_event_create 宏：

```
381 #define switch_event_create(event, id)
        switch_event_create_subclass(event, id, SWITCH_EVENT_SUBCLASS_ANY)
```

以及一个 switch_event_create_subclass 宏：

```
153 #define switch_event_create_subclass(_e, _eid, _sn)
        switch_event_create_subclass_detailed(__FILE__, (const char * )__
SWITCH_FUNC__, __LINE__, _e, _eid, _sn)
```

其中，每一个事件都有一个事件的 ID（一个 switch_event_type_t 的枚举值，对应一个事件名称），以及一个可能的子类型（Subclass）。其中，事件 ID 定义在 switch_types.h:1754 中。只有当事件 ID 为 SWITCH_EVENT_CUSTOM 时子类型才有效。

```
1754     typedef enum {
1755       SWITCH_EVENT_CUSTOM,
1756       SWITCH_EVENT_CLONE,
1757       SWITCH_EVENT_CHANNEL_CREATE,
...
1844       SWITCH_EVENT_ALL
1845   } switch_event_types_t;
```

好了，我们回到 switch_event.c。在调用 switch_event_create_subclass_detailed（第 707 行定义）创建了一个事件的结构之后，可以调用 switch_event_add_header（第 1142 行定义）添加一系列的头数据。这些数据是一些“键 / 值”对。事件中还可以调用 switch_event_add_body（第 1190 行定义）加入一个可选的主体数据。

把所有的事件数据准备好以后，就可以通过 switch_event_fire 将事件发出去了。该函数

实际上是在 switch_event.h:410 中定义的一个宏：

```
410 #define switch_event_fire(event) switch_event_fire_detailed(
        __FILE__, (const char * )__SWITCH_FUNC__, __LINE__, event, NULL)
```

该宏最终映射到 switch_event_fire_detailed 中，它是在 switch_event.c:1944 定义的。它会根据情况在第 1967 行（第一种情况）调用 switch_event_queue_dispatch_event 或在第 1972 行（第二种情况）调用 switch_event_deliver_thread_pool 将事件发送到事件队列中去。

```
1944  SWITCH_DECLARE(switch_status_t) switch_event_fire_detailed(const char
        *file, const char *func, int line, switch_event_t **event, void *user_data)
1945  {
...
1964      if (runtime.events_use_dispatch) {
1967          if (switch_event_queue_dispatch_event(event) != SWITCH_STATUS_SUCCESS) {
1968              switch_event_destroy(event);
1969              return SWITCH_STATUS_FALSE;
1970          }
1971      } else {
1972          switch_event_deliver_thread_pool(event);
1973      }
```

如果是第一种情况，则是在第 383 行将事件推到事件队列中去（旧的方法，为了向后兼容）：

```
383         switch_queue_push(EVENT_DISPATCH_QUEUE, event);
```

如果是第二种情况（默认值），则会使用核心的线程池去分发事件。使用核心线程池进行分发的代码也很简单。首先，在第 276 行申请一块内存，用于存放一个 switch_thread_data_t 结构，然后提供欲在线程池中执行的函数名称（第 280 行）以及一个可选的数据对象指针（第 281 行，在此我们将我们事件的指针作为数据对象的指针传入，然后在第 265 行通过 obj 指针找到传入的事件）。最后在第 286 行从核心线程池中启动一个线程执行第 280 行指定的 switch_event_deliver_thread 函数。

```
272 static void switch_event_deliver_thread_pool(switch_event_t **event)
273 {
274     switch_thread_data_t *td;
275
276     td = malloc(sizeof(*td));
277     switch_assert(td);
278
279     td->alloc = 1;
280     td->func = switch_event_deliver_thread;
281     td->obj = *event;
282     td->pool = NULL;
283
284     *event = NULL;
285
286     switch_thread_pool_launch_thread(&td);
287
288 }
```

switch_event_deliver_thread 函数也非常简单，它只是在第 267 行调用 switch_event_

deliver（该函数我们已经讲过了）将事件分发出去。

```
263 static void *SWITCH_THREAD_FUNC switch_event_deliver_thread(
                                   switch_thread_t *thread, void *obj)
264 {
265     switch_event_t *event = (switch_event_t *) obj;
266
267     switch_event_deliver(&event);
268
269     return NULL;
270 }
```

如果一个模块或线程希望从 FreeSWITCH 事件系统中接收事件，则它应该调用 switch_event_bind_removable 来绑定（订阅）相关的事件。该函数定义在第 1978 行。关于它的内部代码我们就不解释了，后面我们会介绍该函数的使用方法。

```
1978  SWITCH_DECLARE(switch_status_t) switch_event_bind_removable(
        const char *id, switch_event_types_t event, const char *subclass_name,
1979    switch_event_callback_t callback, void *user_data, switch_event_node_t **node)
```

20.3.13 Core Codec 和 Core File

下面我们再来看一下 Core Codec 和 Core File。之所以把两者放到一起讲，是因为他们比较类似——没有太多的业务逻辑，只是对不同的编解码和文件格式的抽象和封装。

在 Core Codec 中，提供了初始化（init）、编码（encode）、解码（decode）、释放（destroy）等函数的抽象，如我们在 20.3.8 节提到的 switch_core_codec_encode 和 switch_core_codec_decode 函数，它们都是在一种编码（codec）与另一种编码（other_codec）间转换。输入参数 decoded_data 表示未编码的（或者说是以 L16 线性编码的）数据缓冲区，而 decoded_data_len 则是数据的长度。同理，encoded_data 和 encoded_data_len 则是编码后的数据缓冲区和长度。如果某种编码有相应的实现代码，则它会向核心注册 codec->implementation->encode 和 codec->implementation->decode 回调函数，所以在下面这两个函数中就直接调用这些回调函数进行编码或解码（如第 736 和第 780 行）。

```
712  SWITCH_DECLARE(switch_status_t) switch_core_codec_encode(switch_codec_t *codec,
713          switch_codec_t *other_codec,
714          void *decoded_data,
715          uint32_t decoded_data_len,
716          uint32_t decoded_rate,
717          void *encoded_data, uint32_t *encoded_data_len,
             uint32_t *encoded_rate, unsigned int *flag)
718  {
...
736      status = codec->implementation->encode(codec, other_codec,
                 decoded_data, decoded_data_len,
737              decoded_rate, encoded_data, encoded_data_len, encoded_rate, flag);

...

744  SWITCH_DECLARE(switch_status_t) switch_core_codec_decode(switch_codec_t *codec,
745          switch_codec_t *other_codec,
```

```
746        void *encoded_data,
747        uint32_t encoded_data_len,
748        uint32_t encoded_rate,
749        void *decoded_data, uint32_t *decoded_data_len,
           uint32_t *decoded_rate, unsigned int *flag)
750  {
...
780       status = codec->implementation->decode(codec, other_codec,
                 encoded_data, encoded_data_len, encoded_rate,
781              decoded_data, decoded_data_len, decoded_rate, flag);
```

与 Core Codec 类似，在 Core File 中，也提供了打开（open）、读（read）、写（write）、关闭（close）等对各种不同的文件操作的抽象。以读和写为例，虽然它的代码与 Core Codec 相比要复杂一些，但基本的功能也是回调各功能模块实现回调函数 fh->file_interface->file_read（如第 295 行或第 324 行）从文件中读取数据到内存缓冲区，或将内存中的数据通过 fh->file_interface->file_write 回调（如第 452 行、第 461 行）写到文件中去。

```
253  SWITCH_DECLARE(switch_status_t) switch_core_file_read(
             switch_file_handle_t *fh, void *data, switch_size_t *len)
254  {
...
295                 if ((status = fh->file_interface->file_read(fh, fh->pre_
                        buffer_data, &rlen)) == SWITCH_STATUS_BREAK) {
...
324         if ((status = fh->file_interface->file_read(fh, data, len)) ==
                SWITCH_STATUS_BREAK) {
...
387  SWITCH_DECLARE(switch_status_t) switch_core_file_write(
                switch_file_handle_t *fh, void *data, switch_size_t *len)
388  {
452        if ((status = fh->file_interface->file_write(fh,
                fh->pre_buffer_data, &blen)) != SWITCH_STATUS_SUCCESS) {
...
461       if ((status = fh->file_interface->file_write(fh, data, len)) ...
```

当然，在 Core File 接口中，除了单纯地读写外还有在缓冲区中对码率的转换以及数据适配等代码，我们就不多讨论了。

总之，在核心中，就是通过这样的抽象与回调机制实现了媒体编解码接口（Codec Interface）、文件接口（File Interface）以及我们前面提到的终点接口（Endpoint Interface），还有我们在本章没有涉及但以前讲过的拨号计划接口（Dialplan Interface）、API 接口（API Interface）、App 接口（Application Interface）等等。

20.4 FreeSWITCH 源代码的编译

关于 FreeSWITCH 的编译，我们在第 3 章中的编译安装中已经提到了。在这里，我们再复习一下，并了解一些开发者需要注意的问题。在此，我们主要以在 Linux 系统上的编译

为例。Windows 平台上的编译过程及注意事项可以参考第 3 章的相关内容。

20.4.1　首次编译

在编译源代码前，请确保安装了第 3 章中所讲的与安装相关的依赖库及开发包。FreeSWITCH 在开发中使用经典的 gcc、Makefile 及 automake、autoconf、libtool 等 GNU 工具链，因而在各种平台上都很容易进行编译。

为了照顾不了解这些 GNU 工具链的读者，我们在此也顺便简单讲一下与 GNU 工具链相关的基础知识。

FreeSWITCH 主要是用 C 和 C++ 编写的。编译 C 语言的程序一般需要 gcc，如下命令会编译 test.c 并生成一个可以执行的二进制程序：

```
gcc test.c -o test
```

当源代码文件数量过多时，一行一行地执行 gcc 就比较累了。因此，可以编写简单的 Shell 脚本或 Makefile 实现。Makefile 是 Make 工具使用的文件，它除了能定义源文件到目标文件的编译方法外，还能定义这些文件的依赖关系。通过检查这些依赖关系，如果在下次编译时源文件没有修改过，则可以不用重复编译，因而可以大大加快编译速度。

不同平台上的工具链是不一样的，在 Linux 等开源平台上一般使用 gcc，而在其他商业的 UNIX 系统上往往都有各厂商自己的编译工具链。为了屏蔽这些不同，一种称为 automake 的工具出现了。通过编写 configure 脚本，定义一些宏，可以在编译前自动检测当前的平台环境和工具链，以生成适当的 Makefile。

当工程更大的时候，写 configure 脚本也是很累人的，因而又有人发明了 automake 和 autoconf，通过定义更简单的宏（m4 宏），可以自动生成 configure 脚本。

总之，通过这些工具和跨平台的宏定义便可以在不同的平台上生成不同的 Makefile，进而可以进行编译。生成 Makefile 的总体流程如图 20-1 所示。读者可以结合 FreeSWITCH 的编译过程深入理解一下，在此我们就不多讲了。

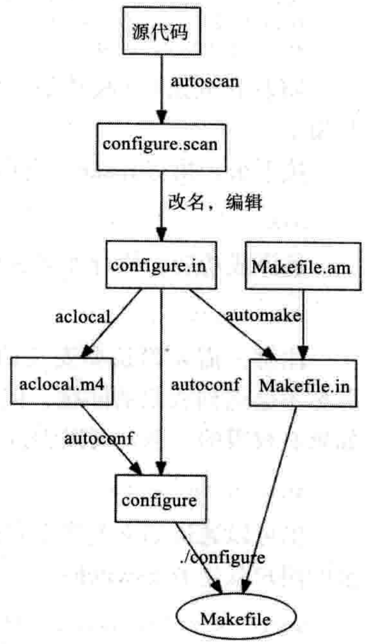


图 20-1　Makefile 的生成过程

如果你是从 Git 仓库中克隆的源代码，要进行编译，则需要先执行一下 bootstrap.sh。它会执行一些初始化操作，生成 configure 文件。

```
./bootstrap.sh
```

如果是直接下载的源代码 Tar 包，则不需要这一步，因为源代码在进行 tar 操作之前就已经执行过该步骤了。

接下来，执行 configure，它会生成 Makefile：

```
./configure
```

configure 有很多参数，其中比较常用的是 prefix 参数，用于将 FreeSWITCH 安装到指定的目录下（FreeSWITCH 默认的安装目录是 /usr/local/freeswitch），如：

```
./configure --prefix=/usr/local/freeswitch2
./configure --prefix=/opt/freeswitch
```

configure 执行完毕后，将产生 Makefile，以及一个 modules.conf 文件。modules.conf 用于控制在编译阶段要自动编译哪些模块。如果你需要这些模块，则可以编辑该文件，并去掉前面的“#”号注释，如：

```
$ head modules.conf
#applications/mod_abstraction
#applications/mod_avmd
#applications/mod_blacklist
#applications/mod_callcenter
#applications/mod_cidlookup
applications/mod_cluechoo
applications/mod_commands
applications/mod_conference
#applications/mod_curl
applications/mod_db
```

如果不知道哪些模块是干什么的，可以暂且不管这个文件。到以后也可以再单独编译某些模块。

接下来，执行 make，它将根据 Makefile 进行编译：

```
make
```

编译成功后，执行如下命令将程序安装到相应的位置：

```
make install
```

注意，需要确认要安装的目标位置有写入的权限，如果这些命令都是以 root 执行的，那你不会遇到权限的问题，但如果你是以普通用户执行的，就可能遇到权限的问题。所以，如果有权限的问题，可以尝试用 root 进行安装：

```
sudo make install
```

也可以通过如下方案以普通用户的身份安装，如以 freeswitch 用户安装，假设你现在登录的用户就是 freeswitch：

```
sudo mkdir /usr/local/freeswitch                    # 用 root 身份创建目录
sudo chown freeswitch /usr/local/freeswitch         # 把目录的属主改为 freeswitch
make install                                        # 用 freeswitch 普通用户身份安装即可
```

20.4.2 增量编译

有时候我们修改了源文件，需要再次编译。在没有修改 autoconf、automake 相关的编译规则的话，直接执行 make 就行了：

```
make
```

也可以直接执行如下命令：

```
make install
```

make 会检查全部的规则，并决定哪些需要重新编译，这还是比较耗时的。如果你知道自己修改了哪些模块，可以直接编译该模块，如：

```
make mod_sofia
make mod_sofia-install
```

使用这种方法也可以编译默认没有编译过的模块，如 mod_shout 模块提供 MP3 录、放音的支持，它默认是不被编译的，可以用以下命令安装：

```
make mod_shout-install
```

当然，在大多数情况下，你也可以直接进入相关的模块目录下，执行 make，如：

```
cd src/mod/endpoints/mod_sofia
make install
```

如果你改了核心的代码，则可以执行如下命令仅编译安装核心部分：

```
make core-install
```

20.4.3 常见问题及最佳实践

如果在编译过程中出现某个或某些模块编译不通过的情况，可以先在 modules.conf 中将该模块注释掉，等全部的编译通过后，再单独检查该模块有什么问题。

如果跟笔者一样，你经常在不同的分支中切来切去，且分支差异比较大时编译系统中的目标文件可能会发生混乱。这可能是编译规则设置的问题，但 FreeSWITCH 项目太大了，因而笔者经常在不同的目录中编译不同的分支，以避免这种混乱。如下列命令编译最新的 master 版本到默认位置：

```
git clone git://git.freeswitch.org/freeswitch.git freeswitch-master
cd freeswitch-master
./bootstrap.sh && ./configure && make && make install
```

编译 1.2 版本并安装到 /usr/local/freeswitch-1.2：

```
git clone git://git.freeswitch.org/freeswitch.git freeswitch-1.2
cd freeswitch-1.2
git checkout v1.2.stable
./bootstrap.sh && ./configure --prefix=/usr/local/freeswitch-1.2
make && make install
```

通过这种方式，以后在维护多个分支时就不会混乱了，而且如果有必要的话，也可以同时在一台主机上同时启动不同版本的 FreeSWITCH 实例（参见 13.1 节）。

20.5 小结

在本章，我们首先讲了 APR 库数据结构、设计理念和原则等相关知识。FreeSWITCH 的源代码依赖于 APR 做跨平台的支持，而且代码风格跟 APR 也非常像，所以很好地了解 APR 对于理解 FreeSWITCH 的源代码是很有帮助的。

在源代码阅读中，我们没有过多地关注有关互斥（Mutex）和锁的代码。而实际上，尤其是对于 FreeSWITCH 这样的多线程的模型的系统来讲，对临界区（多个线程同时访问的资源）加锁是很重要的，而且在使用时一定要非常小心以避免产生竞争条件（Race Condition）或死锁（Deadlock）。本章我们更关注代码的设计理念、封装方式、执行逻辑和流程，使读者在阅读时能很快找到切入点和头绪，以便能更加深入地研究下去。至少具体的数据结构、业务逻辑和相关算法，就留给读者自行研究了。

通过对核心源代码架构的把握，相信读者脑子里已经对系统的整体结构有了一个整体的概念。我们将在下一章给大家深入剖析几种重要的模块实现，并研究一下在外部模块中是怎样使用核心中的功能的。相信通过对下一章的学习，你所了解的所有的知识碎片就都会串到一起了。

除了源代码导读以后，我们在本章还深入讲解了 FreeSWITCH 的编译方法，这样读者就可以在阅读源代码的同时尝试改一些地方（至少，可以在某些关键的地方加一些日志输出的语句），并编译执行看一下效果。阅读永远是枯燥乏味的，只有配合一定的实践，让代码“动”起来，才会有意思。

第 21 章 FreeSWITCH 源代码分析

在上一章，我们一起对 FreeSWITCH 核心源代码进行了简单分析，了解了 FreeSWITCH 源代码的大致结构和流程。在这一章，我们再结合实际的模块对源代码进行更深入的分析，来深入了解这些代码是如何协同工作的。

在上一章，我们带领大家从 main 函数开始看。但有的读者可能还是觉得比较“绕”——一大堆的代码、一大堆的函数调用，很快就把人绕晕了，看了半天还是不得要领。其实这也不怪读者，任何项目、任何代码，从不熟悉到熟悉，总要有一个过程。与代码的作者不同，代码的作者在开发的时候，代码是一行一行写成的，也是一步一步调试成功的，因此整个程序的结构全部在心里。而我们作为一个“外来人”再去看代码，就好像只看一栋盖好的大楼，想去了解其结构和建设过程一样，自然要困难得多。不过，幸运的是我们在上一章已经找到一个突破口——从 main 函数开始大致了解了总的框架结构。本章我们再来找一个突破口，这样一切就比较容易解决了。

21.1 mod_dptools

在此我们找的突破口就是 mod_dptools。该模块包含了系统绝大部分的 App，其中就包括我们熟悉的 answer（应答）和 echo（回声）。不管干什么事情，从熟悉的地方入手，往往比较容易。接下来，我们看一看在源代码里能不能找到 answer 和 echo。

21.1.1 echo

我们先来找 echo。通过使用全文搜索工具搜索源代码，我们很快就在 mod_dptools.c 中找到了一个函数定义——echo_function，它的代码只有下面短短的三行：

```
SWITCH_STANDARD_APP(echo_function)
{
    switch_ivr_session_echo(session, NULL);
}
```

从上述代码中可以看出，echo_function 这个函数是用 SWITCH_STANDARD_APP 宏来定义的。接着跟踪这个宏的定义，发现它是在 switch_types.h:2081 行定义的：

```
2081 #define SWITCH_STANDARD_APP(name) static void name
            (switch_core_session_t *session, const char *data)
```

因此，如果将上面的宏展开，那么 echo_function 的定义就是：

```
static void echo_function(switch_core_session_t *session, const char *data){
    switch_ivr_session_echo(session, NULL);
}
```

我们已经知道，每一路通话（一条腿）均有一个 Session（即这里的 session 变量），每个 App 都是跟 Session 相关的，因而 FreeSWITCH 在调用每个 App 时，均会把当前的 Session 作为参数传入（一个 session 指针）。由于 echo App 没有参数，因而这里的 data 就是空字符串。当然，如果你在 Dialplan 中传入参数，如进行如下操作：

```
<application action="echo" data="some data"/>
```

那么，这里的 char *data 的值就是 some data，只不过我们在此并不需要用到该参数，因而直接忽略掉了。

echo-function 函数就直接调用核心提供的 switch_ivr_session_echo 函数，将收到的 RTP 包原样发回去。switch_ivr_session_echo 函数我们在 20.3.7 节已经详细介绍了，这里就不重复了。

至此，是不是觉得整个呼叫流程一下子就串起来了？当然，如果还是没有这种感觉，我们继续往下看。

继续在 mod_dptools.c 文件中找 echo_function，我们会发现下面一行：

```
SWITCH_ADD_APP(app_interface, "echo", "Echo",
        "Perform an echo test against the calling channel",
echo_function, "", SAF_NONE);
```

它的作用是将我们刚刚定义的 echo_function 加到 app_interface 里（即核心的 Application Interface 指针）。

SWITCH_ADD_APP 也是一个宏，它是在 switch_loadable_modules.c:368 行定义的：

```
368 #define SWITCH_ADD_APP(app_int, int_name, short_descript, \
369       long_descript, funcptr, syntax_string, app_flags) \
370       for (;;) { \
          app_int = (switch_application_interface_t *) \
          switch_loadable_module_create_interface(*module_interface, \
          SWITCH_APPLICATION_INTERFACE); \
371       app_int->interface_name = int_name; \
372       app_int->application_function = funcptr; \
373       app_int->short_desc = short_descript; \
374       app_int->long_desc = long_descript; \
375       app_int->syntax = syntax_string; \
376       app_int->flags = app_flags; \
377       break; \
378 }
```

这个宏定义得非常巧妙，它使用了一个无限的 for 循环，但由于该循环的最后一条语句是 break，因此它只会执行一次。该循环跟 Linux 内核中的“do { ... } while(0)”有异曲同工之妙（参见 20.3.6 节）。

该宏展开后的结果就相当于（为了易读我们去掉了行尾的续行符）：

```
for (;;) {
    app_interface = (switch_application_interface_t *)
        switch_loadable_module_create_interface(*module_interface, SWITCH_APPLICATION_
            INTERFACE);
    app_interface->interface_name = "echo";
    app_interface->application_function = echo_function;
    app_interface->short_desc = "Echo";
    app_interface->long_desc = "Perform an echo test against the calling channel";
    app_interface->syntax = "";
    app_interface->flags = SAF_NONE;
    break;
}
```

所以一个 SWITCH_ADD_APP 相当于使用 switch_loadable_module_create_interface 函数创建了一个 SWITCH_APPLICATION_INTERFACE 类型的接口（即我们所说的 Application Interface）变量 app_interface，然后给它赋予合适的值。大部分参数都是一些描述信息或帮助字符串，最重要的是下面两行，其确定了 echo 这个 app_interface 与我们定义的 echo_function 的对应关系。

```
app_interface->interface_name = "echo";
app_interface->application_function = echo_function;
```

因而，通过 SWITCH_ADD_APP 这个宏，相当于给系统核心添加了一个 echo App，它对应源代码中的 echo_function。这样每当系统执行到 Dialplan 中的 echo App 时，便通过这里的对应关系找到相应的函数入口，进而执行 echo_function 函数。

21.1.2 answer

我们用同样的方法可以找到 answer_function，代码不也不算长，因此我们也把它全部贴在这里：

```
SWITCH_STANDARD_APP(answer_function)
{
    switch_channel_t *channel = switch_core_session_get_channel(session);
    const char *arg = (char *) data;

    if (zstr(arg)) {
        arg = switch_channel_get_variable(channel, "answer_flags");
    }

    if (!zstr(arg)) {
        if (switch_stristr("is_conference", arg)) {
            switch_channel_set_flag(channel, CF_CONFERENCE);
        }
    }
```

```
1224
1225     switch_channel_answer(channel);
1226 }
```

跟 echo_function 类似，该函数也是使用 SWITCH_STANDARD_APP 定义的。我们知道，一个 Session 对应一个 Channel。通过 switch_core_session_get_channel 函数便可以找当前与 Session 对应的 Channel（第 1212 行）。

第 1213 行定义了一个 arg 指针，它指向 answer 的参数 data。如果 arg（即传过来的 data）为空字符串（第 1215 行，zstr 函数用于判断空字符串），则尝试查一下该 Channel 上有没有 answer_flags 这个通道变量，如果有（第 1219 行，其中 switch_stristr 类似于标准的 stristr，不区分大小写），则判断该参数中是否包含“ is_conference”，如果包含，则在该 Channel 上设置一个 CF_CONFERENCE 标志（该标志主要用于 RFC4575/RFC4579 描述的会议系统）。

最后，在第 1225 行调用核心的函数 switch_channel_answer 来对该 Channel 进行应答。switch_channel_answer 函数实际上是一个宏，在此使用一个宏的作用就是往函数中传入调用者的源文件名和行号信息，以便在日志中打印的文件名和行号是实际上调用该函数处的文件名和行号，而不是该函数实际定义处的行号（否则没有什么实际意义）。该宏展开后实际上是调用 switch_channel.c:3693 中的 switch_channel_perform_answer 函数。

在该函数中，它会首先在第 3695 行初始化一个 msg 变量，该变量是 switch_core_session_message_t 类型的，用于定义一条消息（Message）。然后，第 3714 ~ 3715 行初始化消息的内容，并于第 3716 行将消息发送出去，消息（Message）是与 Core Event 类似的另外一种消息传递（调用）方式，与 Core Event 不同的是，消息的发送总是同步进行的，因此这里的 perform_receive_message 实际上是直接调用各模块中接收消息的回调函数，我们在 21.1.3 节还会讲到。

```
3693 SWITCH_DECLARE(switch_status_t) switch_channel_perform_answer(
       switch_channel_t *channel, const char *file, const char *func, int line)
3694 {
3695     switch_core_session_message_t msg = { 0 };
...
3714     msg.message_id = SWITCH_MESSAGE_INDICATE_ANSWER;
3715     msg.from = channel->name;
3716     status = switch_core_session_perform_receive_message(channel->session,
             &msg, file, func, line);
```

如果消息发送成功，就在第 3720 行将该 Channel 的状态置为已经应答的状态。

```
3719     if (status == SWITCH_STATUS_SUCCESS) {
3720         switch_channel_perform_mark_answered(channel, file, func, line);
```

实际上，如果这里的 Channel 是一个 SIP 通话的话，FreeSWITCH 中的 mod_sofia Endpoint 模块便会调用底层的 Sofia-SIP 协议栈（libsofia）给对方发送“200 OK”的 SIP 消息。

answer_function 也是由 SWITCH_ADD_APP 宏安装到核心中去的。

21.1.3 set

FreeSWITCH 中大量使用通道变量控制通话（Channel）的行为。设置通道变量的操作是由下面的 set App 实现的。该函数出奇的简单，因为它直接调用了另外一个函数 base_set。

```
1439 SWITCH_STANDARD_APP(set_function)
1440 {
1441     base_set(session, data, SWITCH_STACK_BOTTOM);
1442 }
```

base_set 其实会被多个函数调用，在此我们只关心它被 set_function 调用的情况。为了更直观，我们通过实际的例子来说明。假设我们在 Dialplan 中使用如下配置：

```
<action application="set" data="dialed_extension=$1"/>
```

其中，$1 为前面的正则表达式的匹配结果，它是一个变量（我们假设它的值为 1001）：

```
1375  static void base_set (switch_core_session_t *session,
                           const char *data, switch_stack_t stack)
1376  {
1377      char *var, *val = NULL;
```

在第 1385 行，会通过 switch_core_session_strdup 将字符串复制一份。该函数是在 session 上进行操作的，它会使用该 session 的内存池申请字符串空间，因而申请以后的内存无需明确释放。为什么要重新复制一份字符串呢？是因为我们接下来的操作会修改该字符串的内存（如第 1392 行），因而复制一份可以避免破坏原来的字符串所占的内存空间。到此，我们的 var 变量的值就是“dialed_extension=$1”了。

```
1385          var = switch_core_session_strdup(session, data);
```

接下来，第 1387 行判断字符串是否包含等号，在我们的例子里有等号，因此 val 指向等号所在的内存位置，也可以说 val 指针所指的字符串值为“=$1”。

```
1387          if (!(val = strchr(var, '='))) {
1388              val = strchr(var, ',');
1389          }
```

如果 val 非空（第 1391 行），则在第 1392 行将 val 所指的位置写入“\0”（即 C 语言中的字符串结束符），并将 val 指针向后移动一个字节，此时它的值就是“$1”了。同时，由于我们将原字符串中的等号替换改成了“\0”，因此，var 所指向的字符串的值也相当于变短了，此时 var 的值为“dialed_extension”。

```
1391          if (val) {
1392              *val++ = '\0';
1393              if (zstr(val)) {
1394                  val = NULL;
1395              }
1396          }
```

第 1398 行，继续判断如果 val 为非空（因为已经移动了指针，所以要重新判断），则执行第 1399 行的函数，就是将 val 指针中的 $1 变量替换为它的实际的值。在这里，我们将在

expanded 变量中得到实际的值是“1001”。

```
1398            if (val) {
1399                expanded = switch_channel_expand_variables(channel, val);
1400            }
```

第 1404 行将调用函数在 Channel 上设置我们指定的新变量：

```
1404            switch_channel_add_variable_var_check(channel,var, expanded,
                    SWITCH_FALSE, stack);
```

它等价于：

```
switch_channel_add_variable_var_check(channel,"dialed_extension", "1001",
    SWITCH_FALSE, stack);
```

当然，最后不要忘记，expanded 指针所指向的内存是动态申请的，因此一定要释放内存，以避免引起内存泄漏。

```
1406            if (expanded && expanded != val) {
1407                switch_safe_free(expanded);
1408            }
```

虽然这里我们讲得比较啰嗦，但实际的过程还是非常简单的。我们接着看一看第 1404 行调用的 switch_channel_add_variable_var_check 函数到底都干了些什么。

该函数定义于 switch_channel.c:1400。它在第 1407 行先对临界区加锁，以防止其他并发的线程同时修改。然后，经过一系列的判断和检查，如果最终所有检查都通过（第 1417 行），则在第 1418 行调用 switch_event_add_header_string 函数将通道变量添加到 channel->variables 中去。该函数与我们在 18.3.1 节讲过的 esl_event_add_header_string 类似，它实际上是向一个 switch_event_t 类型的结构体中添加数据，所以这里可以看到，channel->variables 在内部是使用 switch_event_t 来存储的。这也不奇怪，因为通道变量本来就是一对“键 / 值”对（varname 和 value）。

```
1400  SWITCH_DECLARE(switch_status_t) switch_channel_add_variable_var_check(
              switch_channel_t *channel,
1401          const char *varname, const char *value,
              switch_bool_t var_check, switch_stack_t stack)
1402  {
...
1407      switch_mutex_lock(channel->profile_mutex);
...
1417              if (ok) {
1418                  switch_event_add_header_string(channel->variables,
                          stack, varname, value);
```

当然，永远不要忘了释放锁：

```
1425      switch_mutex_unlock(channel->profile_mutex);
```

至此，set 函数就全部剖析完了。通过它设置的通道变量，以后也可以通过 switch_channel_get_variable 再取出来。当然，这就是另外的事情了。

21.1.4　bridge

接下来我们再来看一下 bridge 这个 App，从某种意义上讲，它属于 FreeSWITCH 的核心功能，比较有代表性。

bridge App 是由第 3017 行的 audio_bridge_function 函数完成的。该函数比较复杂，我们尽量挑简单的部分说。

首先，该 App 在 Dialplan 中的使用方法一般是：

```
<action application="bridge" data="user/1001"/>
```

因而，该函数中的 data 参数便是一个指向字符串“user/1001”的指针。在第 3052 行，首先检查该字符串的有效性。如果它为空字符串，那就没有必要继续进行了，直接返回（return，第 3053 行）即可。

```
3037  SWITCH_STANDARD_APP(audio_bridge_function)
3038  {
...
3052      if (zstr(data)) {
3053          return;
3054      }
```

我们跳过很多 if…else 假设，直接跳到第 3194 行（一般来说都会执行到这里）。接下来的第 3195 行将调用核心的 switch_ivr_originate 函数发起一个新的呼叫。

```
3194          if ((status =
3195               switch_ivr_originate(session, &peer_session, &cause,
                     data, 0, NULL, NULL, NULL, NULL, NULL, SOF_NONE, NULL))
                         != SWITCH_STATUS_SUCCESS) {
3196              fail = 1;
```

switch_ivr_originate 函数是在 switch_ivr_originate.c:1850 行定义的。该函数可能是 FreeSWITCH 中最长的一个函数，在我们参考的版本中，它足足有 2048 行！因此，笔者在此不准备研究它[⊖]，但我们下面来看一下它使用的这几个参数的意义。

switch_ivr_originate 函数中，session 就是指当前的 Session，即呼入的那条腿（a-leg），我们执行到此处，调用该函数创建另一条腿（b-leg）。因而，第二个参数 peer_session 就是新建立的 Session。由于我们在该函数执行完成后，需要知道 peer_session 指针的值，因此这里我们传入的是指针变量的地址（相当于一个双重指针）。同理，我们也需要在呼叫失败时得到呼叫原因（cause），因此把它作为第三个变量。第四个参数便是我们提供的呼叫字符串（data）的指针，在本例中该字符串的值是“user/1001”。

其他的参数我们就没必要看了，大部分都是空指针。在此，由于我们传入了当前的 session 指针，因此该函数在执行的时候就有参照物了。比如，它会将 a-leg（当前 session）

⊖ 这一个函数也许就够写一本书了。好多读者也是在阅读源代码时，好像是在学会了 originate 命令之后，抑或是在知道 bridge App 调用了该函数之后，就来看这个函数。最后的结果要不是认为一个函数写这么长，代码写得太烂；要不就是一下就被吓住了，看不懂。因此，不建议初学者研究这个函数。笔者也是大致看过，并没有深入研究。

中的主叫号码（effective_caller_id_number）作为主叫号码去呼叫 b-leg 等。当然，b-leg 也不是仅参照 a-leg，如果 b-leg 的对端回了呼叫进展消息（如 SIP 180 或 183 消息），则 a-leg 也能听到相关的提示音。

如果 b-leg 的对端应答了，或者在呼叫进展过程中返回了媒体消息（如 SIP 中的 183 消息），则上述的 switch_ivr_originate 函数就会返回。在接下来的第 3207 行，我们将得到新的 Channel（b-leg 对应的 Channel——peer_channel）。

```
3207 switch_channel_t *peer_channel = switch_core_session_get_channel(peer_session);
```

如果我们在呼叫时使用的是 Proxy Media 模式（第 3123 行），则执行第 3214 行的函数时仅进行信令级的桥接，否则（正常情况）就执行第 3236 行的多线程的桥接函数 switch_ivr_multi_threaded_bridge。

```
3213            if (switch_channel_test_flag(caller_channel, CF_PROXY_MODE)) {
3214                switch_ivr_signal_bridge(session, peer_session);
3215            } else {
...
3236                switch_ivr_multi_threaded_bridge(session, peer_session,
                        func, a_key, b_key);
3237            }
```

接下来看 switch_ivr_multi_threaded_bridge 函数。它是在 switch_ivr.c:1270 行实现的。它首先在第 1275 行和第 1276 行始始化了两个 switch_ivr_bridge_data_t 类型的变量 a_leg 和 b_leg，用于存放与两条腿相关的私有数据（我们后面会用到它们）。

```
1270 SWITCH_DECLARE(switch_status_t) switch_ivr_multi_threaded_bridge(
         switch_core_session_t *session,
1271     switch_core_session_t *peer_session,
1272     switch_input_callback_function_t input_callback, void *session_data,
1273     void *peer_session_data)
1274 {
1275     switch_ivr_bridge_data_t *a_leg =
             switch_core_session_alloc(session, sizeof(*a_leg));
1276     switch_ivr_bridge_data_t *b_leg =
             switch_core_session_alloc(peer_session, sizeof(*b_leg));
```

在第 1319 行，它首先在 peer_channel（即 b-leg）上安装一些状态回调函数，当 b-leg 的状态发生变化时，将调用相关的回调函数。

```
1319        switch_channel_add_state_handler(peer_channel, &audio_bridge_peer_
state_handlers);
```

然后产生一个 CHANNEL_BRIDGE 事件（第 1342 行），并发送出去（第 1347 行）。

```
1342          if (switch_event_create(&event, SWITCH_EVENT_CHANNEL_BRIDGE) ==
                SWITCH_STATUS_SUCCESS) {
...
1347              switch_event_fire(&event);
```

接下来，分别在 a-leg 和 b-leg 上产生一个 SWITCH_MESSAGE_INDICATE_BRIDGE 消息（Message，用于标志该 Channel 已经被桥接了），分别发送给这两个 leg（第 1400 行、

第 1408 行）。

```
1396            msg.message_id = SWITCH_MESSAGE_INDICATE_BRIDGE;
...
1400            if (switch_core_session_receive_message(peer_session, &msg)
                    != SWITCH_STATUS_SUCCESS) {
...
1408            if (switch_core_session_receive_message(session, &msg) !=
                    SWITCH_STATUS_SUCCESS) {
```

在第 1439 行，将一个 b_leg 数据指针确定的私有数据绑定到 b-leg 上（使用私有数据与设置通道变量类似，但后者只能是字符串值，而前者可以绑定为任意值）。然后，在第 1440 行将 b-leg 的状态设为媒体交换的状态（CS_EXCHANGE_MEDIA）

```
1439            switch_channel_set_private(peer_channel, "_bridge_", b_leg);
1440            switch_channel_set_state(peer_channel, CS_EXCHANGE_MEDIA);
```

这时候，b-leg 的状态发生了变化，因而会回调在上面第 1319 行设置过的回调函数。不过，在讲这些回调函数前，我们先把第 1442 行讲完。接下来的第 1442 行很简单，它执行 audio_bridge_thread 函数，并将一个 a_leg 数据指针传入。该数据指针包含 a-leg 的一些信息。

```
1442            audio_bridge_thread(NULL, (void *) a_leg);
```

第 1442 行的执行是阻塞的，它将阻塞的执行一直到 bridge 结束，因此，我们可以倒回来看 b-leg 上的回调函数了。

我们在第 1319 行时就在 b-leg 上安装了一些回调函数，这里回调函数是在一个全局变量中指定的，具体如下：

```
771  static const switch_state_handler_table_t audio_bridge_peer_state_handlers = {
772      /*.on_init */ NULL,
773      /*.on_routing */ audio_bridge_on_routing,
774      /*.on_execute */ NULL,
775      /*.on_hangup */ NULL,
776      /*.on_exchange_media */ audio_bridge_on_exchange_media,
777      /*.on_soft_execute */ NULL,
778      /*.on_consume_media */ audio_bridge_on_consume_media,
779  };
```

其中，我们在第 1440 行将 Channel 的状态设置为 CS_EXCHANGE_MEDIA，Channel 的状态发生了改变，它就会回调在第 776 行指定的 audio_bridge_on_exchange_media 函数。

在 audio_bridge_on_exchange_media 函数（第 694 行）中，可以看到，它在第 697 行通过 switch_channel_get_private 取出了该 Channel 上的一个私有数据，而该私有数据是在第 1439 行设置的。该私有数据里面存储了与 bridge 相关的 b-leg 上的数据，有了它以后，我们就可以在第 704 行执行 audio_bridge_thread 函数了。注意，在此第 704 行传入的参数是 b-leg 上的 switch_ivr_bridge_data_t 结构的私有数据。

```
694  static switch_status_t audio_bridge_on_exchange_media(switch_core_
         session_t *session)
695  {
696      switch_channel_t *channel = switch_core_session_get_channel(session);
```

```
697       switch_ivr_bridge_data_t *bd =
             switch_channel_get_private(channel, "_bridge_");
...
703        if (bd->session == session && *bd->b_uuid) {
704            audio_bridge_thread(NULL, (void *) bd);
```

至此，我们可以看到，a-leg 和 b-leg 分别在自己的线程中执行了 audio_bridge_thread 函数（这个很重要，我们的思维现在并行化了，即下面讲的所有代码都是在两条腿上在两个线程中并行执行的），并且在该函数中，它们分别传入了自己所在的那条腿上的 switch_ivr_bridge_data_t 结构的数据。

在该函数中，它首先在第 207 行将传入的数据从 obj 指针赋值给一个 data 指针，并在第 236 行将 data 指针中的 session 成员变量赋值给 session_a。注意，到了这里，session_a 就不一定是 a-leg 了，而是只在当前线程中的那条腿。即，如果在 a-leg 中调用该函数，它就是 a-leg；如果在 b-leg 中调用该函数，它就是 b-leg。同理，第 237 行的 session_b 变量也不一定是 b-leg，而是与本条腿相对的那条腿（桥接中的另一条腿）。注意，该腿相关的 session_b 变量不是直接传入的指针，而是传入了一个 Channel 的 UUID（data->b_uuid），因此我们需要使用 switch_core_session_locate 来取得与该 UUID 对应的 Session 的指针 session_b。

```
205  static void *audio_bridge_thread(switch_thread_t *thread, void *obj)
206  {
207      switch_ivr_bridge_data_t *data = obj;
...
236      session_a = data->session;
237      if (!(session_b = switch_core_session_locate(data->b_uuid))) {
238          return NULL;
239      }
```

至此，在两个线程中分别都有了当前的 Session 和另一个 Session 的信息了，第 256 ~ 257 行分别取出与它们对应的 Channel。然后，就是一个无限循环（第 341 行），在该循环中，不停地在当前的 Session（session_a）中读取一帧媒体数据（第 547 行），然后写入另一个 Session（session_b，第 565 行）。这就实现了媒体[⊖]的交换，也是 bridge App 的全部秘密。当然，第 546 行的注释可能更简洁直观一些——从一个 Channel 中读取音频并写入另一个 Channel。

```
245      chan_a = switch_core_session_get_channel(session_a);
246      chan_b = switch_core_session_get_channel(session_b);
...
341      for (;;) {
...
546          /* read audio from 1 channel and write it to the other */
547          status = switch_core_session_read_frame(session_a, &read_frame,
                 SWITCH_IO_FLAG_NONE, stream_id);
...
565                  if (switch_core_session_write_frame(session_b, read_
                         frame, SWITCH_IO_FLAG_NONE, stream_id) != SWITCH_
                         STATUS_SUCCESS) {
```

⊖ 注意，这里所说的媒体就是音频，为了简单起见，我们没有分析视频有关的代码。

当读取数据错误或者检测到挂机时，上述无限循环将终止。在上述函数中，我们在第 237 行使用了 switch_core_session_locate 通过一个 UUID 获得了 session_b 的指针。而该函数在返回指针的同时会将当前的 Session 加锁，以防止产生竞争条件（Race Condition）。因此，在任何时候使用 switch_core_session_locate 函数并获得了非空的指针时，在指针使用完成后都需要明确解锁，如第 673 行所示：

```
673        switch_core_session_rwunlock(session_b);
```

当然，当上面的 audio_bridge_thread 函数完成后，后续还有很多事情要做，如发送 CHANNEL_UNBRIDGE 事件、检查所有相关的 after_bridge（桥接后的）变量（我们常用的 hangup_after_bridge 变量就是在这里检查的）等，因篇幅所限，我们就不多讲了。

21.1.5 Endpoint Interface

在 mod_dptools 模块中，实现了一些常用的“假”的 Endpoint Interface。之所以说是“假”的，是因为它们并没有像 mod_sofia 那样既有底层的协议驱动，又有媒体收、发处理，而是为了简化某些操作，或者为了在某些特殊的情况下使用一些一致的命令或接口而实现的。比如，我们常用的 user 就是一个 Endpoint。一般来说，一个 Endpoint 都会提供一个用于外呼的呼叫字符串，我们对于 user 提供的呼叫字符串已经非常熟悉了，如在命令行和 Dialplan 中我们经常使用如下的呼叫字符串：

```
originate user/1000 &echo
<action application="bridge" data="user/1000" />
```

这里面的 user 就是由 user Endpoint 实现的。该 Interface 的指针是在第 3879 行声明的一个全局变量。

```
3879  switch_endpoint_interface_t *user_endpoint_interface;
```

在第 3885 ~ 3887 行，定义了一个 switch_io_routines_t 类型的结构体，用于定义回调函数。可以看出，由于该 Endpoint 很简单，它只定义了一个 outgoing_channel 回调函数。该回调函数将在有人使用 user 呼叫字符串时（如执行 originate 和 bridge 时）被调用。

```
3885  switch_io_routines_t user_io_routines = {
3886      /*.outgoing_channel */ user_outgoing_channel
3887  };
```

该回调函数的定义如下：

```
3889  static switch_call_cause_t user_outgoing_channel(switch_core_session_t *session,
3890              switch_event_t *var_event,
3891              switch_caller_profile_t *outbound_profile,
3892              switch_core_session_t **new_session,
              switch_memory_pool_t **pool, switch_originate_flag_t flags,
3893              switch_call_cause_t *cancel_cause)
3894  {
```

首先，该函数的输入参数中将包含一个 outbound_profile，它的成员变量 destination_number 即是被叫号码。在第 3909 行，复制该被叫号码并赋值给 user 指针。

```
3909        user = strdup(outbound_profile->destination_number);
```

然后获取 domain 的值，如果呼叫字符串中未包含 domain（如 user/1000@192.168.1.2 就包含了 domain，而 user/1000 则未包含），则在第 3917 行尝试获取默认的 domain。

```
3914        if ((domain = strchr(user, '@'))) {
3915            *domain++ = '\0';
3916        } else {
3917            domain = switch_core_get_domain(SWITCH_TRUE);
```

接下来，第 3942 行，从 XML 用户目录中查找该用户，并继续在第 3593 行尝试找到 dial-string 配置参数：

```
3942        if (switch_xml_locate_user_merged("id", user, domain, NULL,
            &x_user, params) != SWITCH_STATUS_SUCCESS) {

3953                if (!strcasecmp(pvar, "dial-string")) {
```

如果该呼叫字符串是在 bridge 中使用的，则第 3992 行的判断成立（即说明有 a-leg），否则（第 4004 行）说明是个单腿的呼叫（originate）。然后根据不同的情况进行相关的设置，并得到一个 d_dest（第 4002 行或第 4022 行）的地址（如 sip:1000@192.168.1.100:7890 等）。

```
3992            if (session) {
...
4002                d_dest = switch_channel_expand_variables(channel, dest);
4003
4004            } else {
...
4022                d_dest = switch_event_expand_headers(event, dest);
...
4024            }
```

随后调用 switch_ivr_originate 去呼叫该地址，如第 4040 行所示：

```
4040            } else if (switch_ivr_originate(session,new_session, &cause, d_dest, ...
```

user Endpoint 是一个最简单的 Endpoint。它目前仅支持 SIP 呼叫（理论上它还可以扩展支持其他的），实际的呼叫流程还要转到实际的 mod_sofia Endpoint 上进行处理。我们将在 21.3 节讲解 mod_soifa 是如何实现 Endpoint Interface 的。

21.1.6 模块框架

在 20.3.5 节我们讲过，一个模块主要是由 load、runtime 和 shutdown 回调函数组成的。mod_dptools 当然也不例外。

mod_dptools 模块的 load 函数是在第 5578 行定义的，它将在模块被加载的时候执行。从第 5580 ~ 5584 行可以看出，它实现了包括 API Interface、App Interface、Dialplan Interface 在内的多个 Interface。

```
5578 SWITCH_MODULE_LOAD_FUNCTION(mod_dptools_load)
5579 {
5580     switch_api_interface_t *api_interface;
5581     switch_application_interface_t *app_interface;
```

```
5582       switch_dialplan_interface_t *dp_interface;
5583       switch_chat_interface_t *chat_interface;
5584       switch_file_interface_t *file_interface;
```

在初始化了一系列的内存池及其他数据结构后，它在第 5593 行向核心注册该模块。

```
5593       *module_interface = switch_loadable_module_create_module_
              interface(pool, modname);
```

在第 5595 行，它向核心绑定（订阅）了一个 SWITCH_EVENT_PRESENCE_PROBE 事件的回调函数，即每当系统中产生该事件后，都会执行回调函数 pickup_pres_event_handler。

```
5595       switch_event_bind(modname, SWITCH_EVENT_PRESENCE_PROBE,
           SWITCH_EVENT_SUBCLASS_ANY, pickup_pres_event_handler, NULL);
```

另外，它还实现了一些 EndpointInterface：

```
5621       error_endpoint_interface = ...
5625       group_endpoint_interface = ...
5629       user_endpoint_interface = ...
5633       pickup_endpoint_interface = ...
```

向核心注册 API 和 App：

```
5641       SWITCH_ADD_API(api_interface, "strepoch", ...
5645       SWITCH_ADD_API(api_interface, "strftime", ...
5790       SWITCH_ADD_APP(app_interface, "echo", ...
5791       SWITCH_ADD_APP(app_interface, "park", ...
5794       SWITCH_ADD_APP(app_interface, "playback", ...
```

还可以看到，inline Dialplan 也是在该模块中实现的：

```
5839       SWITCH_ADD_DIALPLAN(dp_interface, "inline", inline_dialplan_hunt);
```

总之，在该函数的最后，返回 SWITCH_STATUS_SUCCESS 表明该模块加载成功：

```
5842       return SWITCH_STATUS_SUCCESS;
```

mod_dptoots 模块没有 runtime 函数。其 shutdown 函数也很简单，该模块没有太多要清理的资源，它只需要在第 5573 行向核心取消先前在第 5593 行绑定的事件回调函数：

```
5571  SWITCH_MODULE_SHUTDOWN_FUNCTION(mod_dptools_shutdown)
5572  {
5573       switch_event_unbind_callback(pickup_pres_event_handler);
5574
5575       return SWITCH_STATUS_SUCCESS;
5576  }
```

上述就是 mod_dptoots 模块的大体框架。读者可以尝试在 FreeSWITCH 控制台上使用 “reload mod_dptools” 命令查看与这些代码相关的日志输出，并对照实际的代码加深一下印象。

21.2 mod_commands

在 mod_commands 中，实现了大部分的 API 命令，如常用的 version、status、originate 等。

21.2.1 模块框架

下面我们先从 mod_commands 模块的 load 函数看起。

load 函数是在第 6388 行定义的，它也是定义了一个 switch_api_interface_t 类型的指针（第 6390 行），用于实现 API Interface，于第 6393 行向核心注册本模块，于第 6420 行、第 6444 行、第 6460 行等向核心注册相关命令实现的回调函数。

```
6388 SWITCH_MODULE_LOAD_FUNCTION(mod_commands_load)
6389 {
6390     switch_api_interface_t *commands_api_interface;
...
6393     *module_interface = switch_loadable_module_create_module_
            interface(pool, modname);
6394
...
6420     SWITCH_ADD_API(commands_api_interface, "version", ...
...
6444     SWITCH_ADD_API(commands_api_interface, "originate", ...
...
6460     SWITCH_ADD_API(commands_api_interface, "status", ...
```

然后，它使用 switch_console_set_complete 添加命令补全信息（如第 6603、6604 行），以便用户在控制台上输入命令时可以使用 Tab 键进行补全。

```
6603     switch_console_set_complete("add show calls");
6604     switch_console_set_complete("add show channels");
```

最后，第 6388 行定义的那个函数返回 SWITCH_STATUS_NOUNLOAD。与其他模块返回 SWITCH_STATUS_SUCCESS 不同，这里的返回值表示该模块是无法被卸载的（由于 unload 命令本身是在该模块内实现的）。

```
6690     return SWITCH_STATUS_NOUNLOAD;
6691 }
```

21.2.2 originate

本节我们以 originate 命令为例来进行讲解。originate 命令是在 originate_function 中实现的。它在第 4067 行使用 SWITCH_STANDARD_API 进行声明。该声明也是一个在 switch_types.h:2016 行定义的一个宏，该宏的定义在笔者的电脑上展开的结果如下：

```
static switch_status_t originate_function ( const char *cmd,
    switch_core_session_t *session, switch_stream_handle_t *stream)
```

从上面的展开结果可以看出，该宏有 3 个输入参数：第一个是输入的命令参数；第二个是一个 session，但由于大多数的 API 命令都跟 Session 无关，因此该参数一般是一个空指针；第三个参数是一个 stream，它是一个流，写入该流中的数据（命令输出）将可作为命令

的结果返回。

第4089行复制了命令字符串。由于originate命令的参数众多，因此它使用一个switch_separate_string对命令字符串进行分隔。该函数将分割后的结果放到一个argv数组中，并返回数组中参数的个数argc(在这一点上，类似于C语言中经典的main函数的参数)。

```
4067  SWITCH_STANDARD_API(originate_function)
4068  {
...
4089      mycmd = strdup(cmd);
4090      switch_assert(mycmd);
4091      argc = switch_separate_string(mycmd, ' ', argv, (sizeof(argv) / sizeof(argv[0])));
```

在对输入参数进行分析后，它便在第4128行调用switch_ivr_originate发起一个呼叫。读者可以看到，bridge App也是调用了该函数发起呼叫，但不同的是，在这里它的第一个参数是一个空指针（NULL），因而这是一个单腿的呼叫。

```
4128      if (switch_ivr_originate(NULL, &caller_session, &cause, aleg,
                timeout, NULL, cid_name, cid_num, ...
```

另外一个与bridge App中调用方法不同的地方在于，在这里它的大部分参数都不是空指针，因而可以在外呼的同时指定其他参数，如超时（timeout)、主叫名称（cid_name)、主叫号码（cid_num）等。

如果我们在发起呼叫时使用“&”指定了一个App，如originate user/1000 &echo命令，则它在对方接听后（严格来说是收到媒体后，如收到SIP 183消息后），即开始执行第4153行，执行app_name（如echo）所指定的函数。

```
4153          if ((extension = switch_caller_extension_new(caller_session,
app_name, arg)) == 0) {
```

否则（第4160行）就在第4161行转移到相应的Dialplan。如用户输入“originate user/1000 9196 XML default”命令则执行下面的代码。

```
4160      } else {
4161          switch_ivr_session_transfer(caller_session, exten, dp,
```

在第4165行，调用输出流stream的write_function输出命令的反馈信息，如“+OK UUID”。

```
4165          stream->write_function(stream, "+OK %s\n",
```

使用switch_ivr_originate所产生的Session也是加锁的，因而，我们也要明确地释放它：

```
4170      switch_core_session_rwunlock(caller_session);
```

21.3 mod_sofia

mod_sofia是FreeSWITCH中最大的一个模块，也是最重要的一个模块。所有的SIP通话都是从它开始和终止的，因而分析该模块的源代码是很有参考意义的。

mod_sofia 模块非常庞大而且复杂，它实现了 SIP 注册、呼叫、Presence、SLA 等一系列的 SIP 特性。在此我们抓住一条主线，仅研究 SIP 呼叫有关的代码，以避免又陷入庞大代码的海洋。

21.3.1 模块加载

我们还是把 mod_sofia 模块的 load 函数作为入口，它是在 mod_sofia.c:5420 实现的。该函数最开始在第 5423 行定义了一个 api_interface 指针，用于往核心中添加 API。在第 5427 行，它将一个全局变量 mod_sofia_globals 清零。该全局变量在整个模块内是有效的，它用于记录一些模块级的数据和变量。然后，在进行一定的初始化后，它在第 5447 行将全局变量的一个 running 成员变量设为 1，标志该模块是在运行的。

```
5420  SWITCH_MODULE_LOAD_FUNCTION(mod_sofia_load)
5421  {
5423      switch_api_interface_t *api_interface;
...
5428      memset(&mod_sofia_globals, 0, sizeof(mod_sofia_globals));
...
5446      switch_mutex_lock(mod_sofia_globals.mutex);
5447      mod_sofia_globals.running = 1;
5448      switch_mutex_unlock(mod_sofia_globals.mutex);
```

在第 5468 行，将启动一个消息处理线程，用于 SIP 消息的处理。

```
5468      sofia_msg_thread_start(0);
```

第 5475 行调用 config_sofia 函数来从 XML 中读取该模块的配置并启动相关的 Sofia Profile。第 5549 行，向核心注册本模块。第 5550 行，初始化一个新的 Endpoint，接着指定该新的 Endpoint 的名字及绑定相关的回调函数（第 5551 行 ~ 第 5554 行）。

```
5475      if (config_sofia(SOFIA_CONFIG_LOAD, NULL) != SWITCH_STATUS_SUCCESS) {
...
5549      *module_interface =
              switch_loadable_module_create_module_interface(pool, modname);
5550      sofia_endpoint_interface =
              switch_loadable_module_create_interface(*module_interface,
                  SWITCH_ENDPOINT_INTERFACE);
5551      sofia_endpoint_interface->interface_name = "sofia";
5552      sofia_endpoint_interface->io_routines = &sofia_io_routines;
5553      sofia_endpoint_interface->state_handler = &sofia_event_handlers;
5554      sofia_endpoint_interface->recover_callback = sofia_recover_callback;
```

后面的代码还有很多，我们就不继续往下看了。至此，我们还有两个细节没有研究明白：第一，就是上面刚刚讲到的这些回调函数都是怎么使用的；第二，就是底层的 Sofia 库是在哪里启动的，又是如何接收 SIP 消息并建立通话的。为了从根本上了解一路通话的建立过程，这次我们先从第二个问题开始看。

21.3.2 Sofia 的加载及通话建立

接下来，我们来看一下 Sofia（即我们的 SIP 服务）到底是从哪里加载的，通话的建立是

从哪里开始的，又是如何进行的。

1. Sofia 的加载

关于 Sofia 的加载，其实我们刚刚已经讲过了，它就隐藏在第 5475 行的 config_sofia 函数中。该函数是在 sofia.c:3585 行定义的。该函数非常长，它解析 XML 配置文件，初始化与 Profile 相关的变量的数据结构，并启动相关的 Profile。我们熟知的默认的 internal Profile 就是在第 4940 行启动的。

```
3585  switch_status_t config_sofia(sofia_config_t reload, char *profile_name)
3586  {
3587      char *cf = "sofia.conf";
...
4940                                launch_sofia_profile_thread(profile);
```

launch_sofia_profile_thread 在第 2817 行定义的，它将于第 2826 行启动一个新线程，并在新线程中执行 sofia_profile_thread_run，同时将 profile 作为输入参数。

```
2817 void launch_sofia_profile_thread(sofia_profile_t *profile)
2818 {
...
2826     switch_thread_create(&profile->thread, thd_attr,
             sofia_profile_thread_run, profile, profile->pool);
2827 }
```

在新线程中（第 2430 行），将在第 2432 行得到 profile 指针的值。然后会在第 2481 行调用 nua_create 函数建立一个 UA（User Agent）。nua_create 是 Sofia-SIP 库提供的函数，它将启动一个 UA，监听相关的端口（如大家熟知的 5060），并等待 SIP 消息到来。一旦收到 SIP 请求，它便会回调 sofia_event_callback 回调函数（第 2482 行），该回调函数中将带着对应的 profile 作为回调参数。

```
2430 void *SWITCH_THREAD_FUNC sofia_profile_thread_run(
                                  switch_thread_t *thread, void *obj)
2431 {
2432     sofia_profile_t *profile = (sofia_profile_t *) obj;
...
2481    profile->nua = nua_create(profile->s_root,   /* Event loop */
2482        sofia_event_callback, /* Callback for processing events */
2483        profile,  /* Additional data to pass to callback */
2484        ...
```

关于 Sofia-SIP 底层的库我们就不深入研究了。到此为止，我们的 SIP 服务已经启动了，就等着接收 SIP 消息了。

2. SIP 消息的接收

当我们的服务收到 SIP 消息后，便会调用 sofia_event_callback 回调函数。该函数在第 1789 行定义。在该行，如果得到回调，则收到一个 nua_event_t 结果的 SIP 事件 event。即使不看 Sofia-SIP 库的文档，我们也能从第 1800 行的 switch 语句以及后面的 case 分支中看出——该事件到底是对应什么类型的 SIP 消息了。如果收到 SIP INVITE 消息，那么它一定

会匹配到第 1840 行。

```
1789  void sofia_event_callback(nua_event_t event,
1790      int status,
1791      char const *phrase,
1792      nua_t *nua, sofia_profile_t *profile,
    nua_handle_t *nh, sofia_private_t *sofia_private,
    sip_t const *sip,
1793      tagi_t tags[])
1794  {
...
1800      switch(event) {
1801      case nua_i_terminated:
...
1840      case nua_i_invite:
1841      case nua_i_register:
1842      case nua_i_options:
1843      case nua_i_notify:
1844      case nua_i_info:
```

不同的消息将进行不同的处理，但大部分都会执行到第 2018 行，将消息通过一个核心的消息队列分发出去（其中，de 是一个 sofia_dispatch_event_t 的结构体指针，它包含了本次收到的 SIP 消息）。

```
2018      sofia_queue_message(de);
```

Sofia-SIP 库在底层是一个单线程的结构，因此在这里我们使用了消息队列以提高并发量。

接下来我们跟踪到第 1749 行，sofia_queue_message 函数将保证我们的消息队列有足够的处理能力。然后，进行必要的检查。如果这是第一次 INVITE 请求（第 1881 行），则在第 1943 行（略）或第 1945 行调用 switch_core_session_request 生成一个新的 Session，并赋值给 session 指针。到这里，INVITE 请求在我们系统中起作用了——它导致我们的系统创建了一个 Session，在以后所有与该 INVITE 消息相关的会话消息中，都会与该 Session 相关（当然，具体的关联代码还有很多，我们就不再深入了）。

```
1749 void sofia_queue_message(sofia_dispatch_event_t *de)
1750 {
...

1881      if (event == nua_i_invite && !sofia_private) {
...
1945          session = switch_core_session_request(sofia_endpoint_interface,
SWITCH_CALL_DIRECTION_INBOUND, SOF_NONE, NULL);
```

接下来，在第 1950 行，初始化一个 tech_pvt 指针（该指针所指向的结构中将用于保存本 Session 的私有数据，我们后面还会讲到）。在第 1962 行，将这些私有数据与 session 绑定。然后，在第 1981 行，为该 Session 启动一个新的线程，以执行后续耗时的操作，避免阻塞当前的线程。

```
1950          tech_pvt = sofia_glue_new_pvt(session);
...
1962          sofia_glue_attach_private(session, profile, tech_pvt, channel_name);
```

```
...
1981            if (switch_core_session_thread_launch(session) != SWITCH_STATUS_SUCCESS) {
```

当然，启动了新线程后，对该 SIP 事件的处理还没有完，它还会后续设置 de 指针，并将收到的消息通过第 1777 行的 switch_queue_push 推到一个模块级的消息队列中去（即这里的 msg_queue）。

```
2003            de->init_session = session;
...
1777        switch_queue_push(mod_sofia_globals.msg_queue, de);
1778 }
```

至此，SIP 事件的接收就完成了。如果后续收到其他 SIP 事件，将进行下次回调，并推到队列中等候处理。

3. SIP 消息的处理

对 SIP 事件的处理是在单独的线程（组）中执行的。进行事件处理的线程是在模块加载时从 sofia_msg_thread_start 函数开始的。该函数定义于 sofia.c:1715，它首先会启动一个新线程，并在以后根据 CPU 的数量以及当前的需要决定启动多少个消息处理线程。从第 1738 行可以看出，新的事件处理线程中将执行 sofia_msg_thread_run 函数。

```
1715 void sofia_msg_thread_start(int idx)
1716 {
...
1736        switch_thread_create(&mod_sofia_globals.msg_queue_thread[i],
1737                                      thd_attr,
1738                                      sofia_msg_thread_run,
```

我们继续跟踪，就发现 sofia_msg_thread_run 是在第 1671 行定义的。它在第 1691 行使用一个无限循环，不断地从消息队列中取出一条消息（事件），然后在第 1700 行使用 sofia_process_dispatch_event 函数发送出去。

```
1671 void *SWITCH_THREAD_FUNC sofia_msg_thread_run(
                                        switch_thread_t *thread, void *obj)
1672 {
...
1691        for(;;) {
1692
1693            if (switch_queue_pop(q, &pop) != SWITCH_STATUS_SUCCESS) {
...
1699                sofia_dispatch_event_t *de = (sofia_dispatch_event_t *) pop;
1700                sofia_process_dispatch_event(&de);
```

看来还得继续跟踪，sofia_process_dispatch_event 被定义在第 1643 行，当看到第 1652 行时，我们总算看到了一点曙光，它终于像在调用一个回调函数了。

```
1643 void sofia_process_dispatch_event(sofia_dispatch_event_t **dep)
1644 {
...
1652        our_sofia_event_callback(de->data->e_event, de->data->e_status,
                de->data->e_phrase, de->nua, de->profile,
1653            de->nh, sofia_private, de->sip, de, (tagi_t *)de->data->e_tags);
```

继续往下跟踪可以确定我们的猜测，在第 1024 行找到 our_sofia_event_callback 的定义后，可以看到它确实是在处理 SIP 消息了。在第 1130 行的 switch 语句的各个分支中，我们可以看到许多以 nua_r_ 和 nua_i_ 开头的 SIP event，其中，前者表示收到一条响应（Response）消息，而后者表示收到一条请求消息。

```
1024  static void our_sofia_event_callback(nua_event_t event,
1025      int status,
1026      char const *phrase,
1027      nua_t *nua, sofia_profile_t *profile, nua_handle_t *nh,
        sofia_private_t *sofia_private, sip_t const *sip,
1028      sofia_dispatch_event_t *de, tagi_t tags[])
1029  {
...
1130      switch (event) {
1131      case nua_r_get_params:
1132      case nua_i_fork:
```

我们集中精力看 INVITE 消息，如果收到 INVITE 消息，则第 1247 行的 case 条件成立。继续判断如果第 1249 行的条件成立，则说明是一个 re-INVITE 消息，在第 1250 行进行处理。否则，则说明是一个新的 INVITE 消息，在第 1253 行调用 sofia_handle_sip_i_invite 处理。

```
1247      case nua_i_invite:
1248          if (session && sofia_private) {
1249              if (sofia_private->is_call > 1) {
1250                  sofia_handle_sip_i_reinvite(...
1251              } else {
1252                  sofia_private->is_call++;
1253                  sofia_handle_sip_i_invite(session, nua, profile, nh,
                        sofia_private, sip, de, tags);
1254              }
1255          }
```

在 sofia_handle_sip_i_invite 中，将更深入解析 INVITE 消息，对 Session 的相关内容进行更新，如果需要对来话进行认证，还需要给对方发送 SIP 407 消息进行挑战认证等。sofia_handle_sip_i_invite 函数是在第 7845 行定义的，有兴趣的读者可以找到它来研究一下，在此我们就不再深入了。

4. SIP 状态机

在 Sofia-SIP 底层，也实现了一个状态机，在 SIP 通话的不同阶段使用不同的状态进行表示和处理。因而在 SIP 状态发生改变时，它便向上层上报状态变化事件，这些状态变化事件也是在 SIP 事件的形式上报的，因而会经过跟上述的 INVITE 消息类似的回调过程一直到回调同一个回调函数 our_sofia_event_callback。在该函数的第 1265 行，我们会看到在收到 Sofia-SIP 底层驱动的状态变化后，继续回调 sofia_handle_sip_i_state 函数来进行相关处理。

```
1265      case nua_i_state:
1266          sofia_handle_sip_i_state(session, status, phrase, nua,
                  profile, nh, sofia_private, sip, de, tags);
```

由于 sofia_handle_sip_i_state 函数有太多的状态和情况需要处理，因此也非常长。我们

很难通过直接阅读源码的方式找到正确的入口。看起来，虽然我们代码剖析到最后了，但是竟然可能要迷失了。

不过，办法总比困难多。在本章中，我们过多地关注了理论却少了实践。下面让我们拿起一个 SIP 电话，拨打 9196，很快就可以在日志中看到如下的信息：

```
[DEBUG] sofia.c:5861 ... Channel entering state [received][100]
```

从上一条日志可以看出，在第 5861 行打印了一条日志，表示我们的状态机进入了收到 INVITE 消息后发送 100 Trying 消息的阶段（代码略）。而接着下一条日志则告诉我们 Channel 的状态从 CS_NEW 变成了 CS_INIT。

```
[DEBUG] sofia.c:6116 ... State Change CS_NEW -> CS_INIT
```

有了上述信息，我们就可以在 sofia.c 的第 6116 行很快找到该日志对应的代码了。只要满足一定的条件，在该行就会把 Channel 的状态变为 CS_INIT，然后 Channel 的核心状态机就会回调相关的状态回调函数了。

```
5773  static void sofia_handle_sip_i_state(...
5778  {
...
6115                      if (switch_channel_get_state(channel) == CS_NEW) {
6116                          switch_channel_set_state(channel, CS_INIT);
```

5. Channel 状态机

只要 Channel 的状态一变成 CS_INIT，FreeSWITCH 核心的状态机代码就会负责处理各种状态变化了，因而各 Endpoint 模块就不需要再自己维护状态机了。也就是说，在一个 Endpoint 模块，首先要有一定的机制用于初始化一个 Session（对应一个 Channel，它的初始状态将为 CS_NEW），然后在适当的时候把该 Channel 的状态变成 CS_INIT，剩下的事就基本不用管了。

当然，这里说的是“基本”而不是“绝对”。一般来说，还是要在 Endpoint 模块中跟踪 Channel 状态机的变化，这就需要靠在核心状态机上注册相应的回调函数实现，如 Sofia Channel 的状态机的回调是在第 4012 行定义的。

```
4012  switch_state_handler_table_t sofia_event_handlers = {
4013      /*.on_init */ sofia_on_init,
4014      /*.on_routing */ sofia_on_routing,
4015      /*.on_execute */ sofia_on_execute,
4016      /*.on_hangup */ sofia_on_hangup,
4017      /*.on_exchange_media */ sofia_on_exchange_media,
4018      /*.on_soft_execute */ sofia_on_soft_execute,
4019      /*.on_consume_media */ NULL,
4020      /*.on_hibernate */ sofia_on_hibernate,
4021      /*.on_reset */ sofia_on_reset,
4022      /*.on_park */ NULL,
4023      /*.on_reporting */ NULL,
4024      /*.on_destroy */ sofia_on_destroy
4025  };
```

这些回调函数是在 21.3.1 节讲过的代码的第 5553 行注册到核心中去的。回调函数本身

都比较简单，感兴趣的读者可以自己看一下代码。为了节省篇幅，在此我们就不多列举了。

6. IO 例程

与 Channel 状态机回调相比，Endpoint 模块中更重要的是 IO 例程的回调。IO 例程主要提供媒体数据的输入输出（IO）功能。与上一节讲的 Channel 的状态机类似，IO 例程的回调函数是在 21.3.1 节讲到的代码的第 5552 行注册到核心中去的。其中，IO 例程的回调函数是由一个 switch_io_routines_t 类型的结构体变量设置的，该变量的定义在第 3997 行。

```
3997  switch_io_routines_t sofia_io_routines = {
3998      /*.outgoing_channel */ sofia_outgoing_channel,
3999      /*.read_frame */ sofia_read_frame,
4000      /*.write_frame */ sofia_write_frame,
4001      /*.kill_channel */ sofia_kill_channel,
4002      /*.send_dtmf */ sofia_send_dtmf,
4003      /*.receive_message */ sofia_receive_message,
4004      /*.receive_event */ sofia_receive_event,
4005      /*.state_change */ NULL,
4006      /*.read_video_frame */ sofia_read_video_frame,
4007      /*.write_video_frame */ sofia_write_video_frame,
4008      /*.state_run*/ NULL,
4009      /*.get_jb*/ sofia_get_jb
4010  };
```

在 21.1.5 节，我们已经讲过了 outgoing_channel 的回调，该回调是在有外呼请求的时候（如，执行“originate sofia/gateway/...”时）被回调执行的。在此我们再来看一下 mod_sofia 中的 outgoing_channel 有何不同。

该模块的 outgoing_channel 回调是在第 4032 行定义的。在第 4057 ~ 4058 行，它也是初始化了一个新的 Session（nsession），然后初始化了一个新的 tech_pvt 用于存放私有数据。第 4065 行还是从 outbound_profile 中复制被叫号码，第 4071 行得到对应的 Channel（nchannel）。如果该外呼是由 bridge 发起的，则还会有 a-leg 存在，因而在第 4074 行将得到与 a-leg 对应的 Channel，我们新生成的 nchannel 即是 b-leg。

```
4032  static switch_call_cause_t sofia_outgoing_channel(...
...
4057      if (!(nsession = switch_core_session_request_uuid(
                sofia_endpoint_interface, SWITCH_CALL_DIRECTION_OUTBOUND,
4058            flags, pool, switch_event_get_header(var_event,
                                 "origination_uuid")))) {

4063      tech_pvt = sofia_glue_new_pvt(nsession);
4064
4065      data = switch_core_session_strdup(nsession,
                     outbound_profile->destination_number);
...
4071      nchannel = switch_core_session_get_channel(nsession);
4072
4073      if (session) {
4074          o_channel = switch_core_session_get_channel(session);
4075      }
```

之后，在第 4346 行将 tech_pvt 与 nsession 关联进来。

```
4346        sofia_glue_attach_private(nsession, profile, tech_pvt, dest);
```

从第 4428 ~ 4430 行可以看出，新的 Channel nchannel 的状态变为了 CS_INIT。然后，该 Channel 便进入正常的呼叫流程了。接下来，核心的状态机会接管后面的状态变化，如将状态机置为 CS_ROUTING，然后进行路由查找（即查找 Dialplan），最后进入 CS_EXECUTE 状态，执行在 Dialplan 中找到的各种 App 等。

```
4428        if (switch_channel_get_state(nchannel) == CS_NEW) {
4429            switch_channel_set_state(nchannel, CS_INIT);
4430        }
```

当代码中某处调用 switch_core_session_read_frame 试图读取一帧音频数据时（如 21.1.4 节中的情况），就会执行 read_frame 回调函数，因而会回调第 929 行定义的函数。该回调函数由于将大部分功能都移动到核心的 Core Media 代码中去了，因而非常简单，它主要就是在第 964 行调用核心的 switch_core_media_read_frame 从底层的 RTP 中读取音频数据。

```
929  static switch_status_t sofia_read_frame(switch_core_session_t *session,
        switch_frame_t **frame, switch_io_flag_t flags, int stream_id)
930  {
...
964      status = switch_core_media_read_frame(session, frame, flags,
                 stream_id, SWITCH_MEDIA_TYPE_AUDIO);
...
968      return status;
969  }
```

当然，写数据的情况与此差不多，write_frame 回调函数在第 971 行定义。它在第 1008 行也是调用了 Core Media 中的函数 switch_core_media_write_frame 通过 RTP 将音频数据发送出去。

```
971  static switch_status_t sofia_write_frame(switch_core_session_t *session,
        switch_frame_t *frame, switch_io_flag_t flags, int stream_id)
972  {
...
1008        if (switch_core_media_write_frame(session, frame, flags,
                stream_id, SWITCH_MEDIA_TYPE_AUDIO)) {
```

视频的回调函数 read_video_frame 和 write_video_frame 与此差不多。我们就不多讲了。最后，还有一个比较有意思的 receive_message 回调，看第 1096 行定义的回调函数。其中，在第 1123 行和第 1244 行各一个 switch 语句用于判断收到的各种消息，并进行相应的处理。我们直接跳到第 2031 行，在收到 SWITCH_MESSAGE_INDICATE_ANSWER 消息时，它将在第 2032 行调用 sofia_answer_channel 对当前通话进行应答。

```
1096  static switch_status_t sofia_receive_message(switch_core_session_t *session,
                                switch_core_session_message_t *msg)
1097  {
...
1123        switch (msg->message_id) {
...
```

```
1244      switch (msg->message_id) {
...
2031      case SWITCH_MESSAGE_INDICATE_ANSWER:
2032          status = sofia_answer_channel(session);
```

可很容易想象到，应答将会向对方发送 SIP 200 OK 消息。而它就是在第 608 行调用 Sofia-SIP 底层库实现的。

```
602 static switch_status_t sofia_answer_channel(switch_core_session_t *session)
603 {
...
608                   nua_respond(tech_pvt->nh, SIP_200_OK, ...
```

如果我们再回到 21.1.2 节的 answer App，就可以看到它调用了核心的 switch_answer_channel 函数，并在 switch_channel.c:3716 行发送了一个 SWITCH_MESSAGE_INDICATE_ANSWER 消息，因而 sofia_receive_message 函数被回调，代码执行到了第 2032 行，并最终在第 608 行向对方的 SIP 终端发送 200 OK 消息。

至此，我们所有的呼叫流程就全部都串起来了，我们对源代码的分析也到此结束了。

21.4 小结

在本章，我们对 3 个最有代表性的模块的源代码进行了深入剖析。结合上一章所学的内容，以我们最熟悉的 echo、answer 等 App 作为突破口，一步一步深入跟踪，终于理清了代码的执行流程，了解了各种回调函数的含义及触发时机。同时，我们也对从模块启动、网络监听、来话的接收、Session 的生成、各种状态的转移直到应答等全部的流程进行了跟踪和梳理。

通过对本章的学习再配合系统的运行日志，便可轻松深入地学习和研究源代码了。当然，FreeSWITCH 的代码非常多，学习源代码更多的是需要细心和耐心。退一万万步讲，这些代码是用了将近十年的时间写成的，不要指望一天就能精通。另外，笔者也不可能在短短几章内把所有的概念、方法、流程讲清楚。掌握 FreeSWITCH 的代码还需要阅读代码、多实践、多调试，总之，多下工夫才会有更多收获。

第 22 章 FreeSWITCH 二次开发

通过对本书前面章节的学习，我们熟悉了 FreeSWITCH 的使用方法，熟悉了 FreeSWITCH 的源代码。现在，该轮到我们自己写代码的时候了。当然，在前面我们也写过代码，但都是使用嵌入式脚本及 ESL 接口等在 FreeSWITCH 外部开发的。在本章，我们将以笔者在学习和使用过程中的实际例子为例，讲一下如何修改 FreeSWITCH 的源代码，如何将自己的成果提交到官方的代码库里，为开源项目做贡献。最后，带领大家从头开始开发一个新的模块，以复习和巩固前面学过的知识，告别“纸上谈兵”，冲到战场的一线，为将来的开发打下坚实的基础。

22.1 给 FreeSWITCH 汇报 Bug 和打补丁

大多数商业的系统都是闭源的，有时候出了问题甚至很难跟踪调试，更不容易发现 Bug 的具体位置。而相对来讲，使用开源项目的好处就是，我们可以参照源代码比较容易找到 Bug。笔者在学习和使用 FreeSWITCH 的过程中，在官方的 Bug 跟踪工具中汇报了很多的 Bug [⊖]，大多数都得到了很及时的修复。接下来，我们就一起看几个真实的例子（为节省篇幅，本章部分代码使用 git diff 格式，其中，行前的“-”表示删除的行，“+”代表添加的行）。

22.1.1 修复内存泄露问题

在笔者早期使用 FreeSWITCH 的过程中，发现 FreeSWITCH 的内存一直增长。查找了很久没有找到问题原因。而与同时，笔者却没有发现其他人遇到同样的问题。后来，笔者考虑到自己使用了 mod_erlang_event 模块，而使用该模块的人比较少，因此就研究了一个该

⊖ 当然，不要因此误认为开源的系统 Bug 多，其实闭源的通常 Bug 更多，只是你看不见而已。因为开源的项目，全世界的人都在各种应用场景下使用，所有人都能看到源代码，因而更容易发现 Bug，而且通常开源社区也会更及时地修复。

模块的源代码（当时还不会使用 valgrined 工具查找内存泄露问题）。终于发现一处申请了内存似乎没有释放。后来，在源代码中，尝试增加了如下一行代码，在 event 指针用完之后将内存释放。

```
switch_event_destroy(&event);
```

进行了简单的压力测试后，发现问题似乎解决了。又在生产系统上运行了几天之后，确认没有问题了，笔者便在官方的 Bug 跟踪系统（Jira）上提交了一个 Issue 描述了发现的问题。然后，使用 git diff > erlang_leak.diff 命令产生了下面的补丁文件，并将补丁文件也附加了上去。

```
diff --git a/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
    b/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
index 9a09e80..cd58d95 100644
--- a/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
+++ b/src/mod/event_handlers/mod_erlang_event/mod_erlang_event.c
@@ -650,6 +650,9 @@ static switch_status_t check_attached_sessions(listener_t *listener)
    }

    switch_thread_rwlock_unlock(listener->session_rwlock);
+
+   switch_event_destroy(&event);
+
    if (prefs.done) {
        return SWITCH_STATUS_FALSE; /* we're shutting down */
    } else {
```

最后，该模块的作者将补丁合并到 FreeSWITCH 代码库中进去了。相关的 Jira 参见：http://jira.freeswitch.org/browse/FS-3488。

22.1.2 给中文模块打补丁

由于我们需要在 FreeSWITCH 中支持中文语音，因此要用到 mod_say_zh 模块。而在使用的过程中我们的团队成员发现了如下的错误：

```
[ERR] mod_say_zh.c:513: Unknown Say type=[18]
```

笔者鼓励同事去源代码里找一找出错的原因，他们很快就找到它是在第 513 行的一条日志输出语句中输出的。从源代码可以看到，错误的原因很明显，就是在 case 语句中没有对应的分支，进而转到 default 语句造成的。相关的部分代码片断如下：

```
490 case SST_NUMBER:
...
511 case SST_CURRENCY:
...
512 default:
513      switch_log_printf(SWITCH_CHANNEL_LOG, SWITCH_LOG_ERROR,
         "Unknown Say type=[%d]\n", say_args->type);
```

通过在全部源代码中搜索离它最近的第 511 行的常量定义 SST_CURRENCY，我们找到了对应错误日志中 18 的是常量 SST_SHORT_DATE_TIME（switch_types.h:418）。更深入的

研究发现其实只需对它做与 SST_CURRENT_DATE_TIME 同样的处理即可。因而，我们产生了一个补丁。由于当时笔者已经具有代码库的提交权限，因此就直接将代码提交到了代码库中。而没有提交 Jira。

如果读者有源代码的话，可以用以下命令查看相关的补丁（命令和输出结果如下，为节省篇幅，输出结果有删节）：

```
$ git show f255f6
commit f255f65a82e2cfe1aed1f44aa2459e5faaed150e
Author: Seven Du <dujinfang@gmail.com>
Date:   Thu Aug 1 09:50:51 2013 +0800

    add SHORT_DATE_TIME support

diff --git a/src/mod/say/mod_say_zh/mod_say_zh.c ...
        case SST_CURRENT_DATE:
        case SST_CURRENT_TIME:
        case SST_CURRENT_DATE_TIME:
+       case SST_SHORT_DATE_TIME:
                say_cb = zh_say_time;
```

通过这次修复，再使用该模块就不会再遇到这个错误了。

22.1.3　给 FreeSWITCH 核心打补丁

笔者在撰写本书的时候，阅读了大量的源代码，偶然发现其中对于多个 Channel 进行混音的源代码中可能有问题。该段代码不长，因此我们把它们全部贴在后面。

该函数的作用是对音频数据进行混音，即将多个声道的数据合并到一个声道里。

对多个 Channel 进行混音的函数是在第 274 行定义的，所有的音频数据存放到 data 指针中。其中，音频数据是 16 位的整型数据（int16_t）；samples 代表采样率，如 8000；channels 代表有几个声道，如果在双声道中，它的值就是 2。

第 276 行，定义了一个 buf 指针，备用；第 277 行，算出需要的缓冲区的字节长度；第 279 行，定义 32 位的无符号整型变量（uint32_t），以避免在计算中 16 位的整数溢出。

```
274  SWITCH_DECLARE(void) switch_mux_channels(int16_t *data,
         switch_size_t samples, uint32_t channels)
275  {
276      int16_t *buf;
277      switch_size_t len = samples * sizeof(int16_t);
278      switch_size_t i = 0;
279      uint32_t j = 0, k = 0;
```

第 281 行，申请一个足够大的缓冲区，让 buf 指针指向它。然后使用一个双重 for 循环将两个声道对应位置的数据相加（第 285 行），并于第 286 行使用 switch_normalize_to_16bit ⊖ 将 32 位的整数标准化成 16 位的整数，在第 287 行将数据写入缓冲区。

```
281      switch_zmalloc(buf, len);
282
```

⊖ 看起来像一个函数，实际上是一个在 switch_utils.h:236 定义的宏。

```
283        for (i = 0; i < samples; i++) {
284            for (j = 0; j < channels; j++) {
285                int32_t z = buf[i] + data[k++];
286                switch_normalize_to_16bit(z);
287                buf[i] = (int16_t) z;
288            }
289        }
```

全部处理完成后，再使用 memcpy 内存拷贝函数将数据从缓冲区中复制到原来的数据区域（第 291 行），并释放缓冲区（第 292 行）。

```
291        memcpy(data, buf, len);
292        free(buf);
293
294    }
```

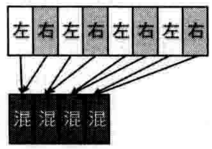


图 22-1 双声道混音

具体的算法应该很直观。一个双声道混音的示意图如图 22-1 所示（其中“左”、“右”分代表左、右声道）。

在笔者刚刚看到该代码时，也是花了一些时间明白了混音的算法。不过，笔者立即想到，既然是混音，那么如果将两个声道混合成一个声道，理论上只占用一半的内存缓冲区，所以原来的缓冲区如果可以重复利用的话，能不能不申请新的缓冲区呢？通过一番实验，笔者实现了如下改进的算法。

在新的算法中，笔者还是使用一个双重 for 循环遍历所有数据，但是，这里没有申请新的内存缓冲区，而是在第 280 行使用了一个新的中间变量 z。将对应的音频数据相加后放到 z 中（第 282 行），然后将数据标准化（第 283 行），并直接将最后结果写入原来的 data 指针所传入的数据缓冲区即可（第 284 行，因为原来的数据我们已经取出来计算过了，不再需要了）。

```
274    SWITCH_DECLARE(void) switch_mux_channels(int16_t *data,
           switch_size_t samples, uint32_t channels)
275    {
276        switch_size_t i = 0;
277        uint32_t j = 0;
278
279        for (i = 0; i < samples; i++) {
280            int32_t z = 0;
281            for (j = 0; j < channels; j++) {
282                z += data[i * channels + j];
283                switch_normalize_to_16bit(z);
284                data[i] = (int16_t) z;
285            }
286        }
287    }
```

通过使用新的算法，至少节省了一半的内存，而且避免了重复的内存申请造成的内存碎片，节省了一个内存拷贝操作及 6 行代码。笔者并没有实际测试以比较最终的效果，不过理论上是这样的。

在经过反复测试确认没有问题之后，笔者将该改进提到了官方的 Jira 上，见：http://jira.freeswitch.org/browse/FS-4622。虽然笔者当时具有直接向代码库中提交代码的权限，但是毕竟对于核心代码的改动问题比较重大。把它记录到 Jira 上，以后万一出了问题也容易跟踪。

而且，Jira 会给原代码的作者一个比较友好的通知，让作者决定是否将代码合并进去，这是一种比较友好的协作方式。

22.1.4 高手也会犯错误

在 20.3.2 章我们讲到过，由于 SWITCH_CHANNEL_SUCCESS 的常量值为 0，因此，在使用时需要严格进行“==”判断，否则就容易出现错误。在 FreeSWITCH 代码的历史上，就曾经出现过这样的错误，其中一次是笔者发现的，记录在该 Jira 报告中：http://jira.freeswitch.org/browse/FS-5351。

该 Bug 已经修复，感兴趣的读者仍可以使用 git show c4e7c30 命令查看当时是如何修复的，以避免自己以后犯同样的错误。如下可以看出，在“+”一行，增加了“SWITCH_STATUS_SUCCESS ==”判断。

```
    $ git show c4e7c30
    ...
    diff --git a/src/mod/endpoints/mod_sofia/mod_sofia.c ...
    @@ -968,7 +968,7 @@ static switch_status_t sofia_write_video_frame(switch_core_
session_t *session, s
                    return SWITCH_STATUS_SUCCESS;
            }

    -          if (switch_core_media_write_frame(session, frame, flags, stream_id,
SWITCH_MEDIA_TYPE_VIDEO)) {
    +       if (SWITCH_STATUS_SUCCESS == switch_core_media_write_frame(
    +           session, frame, flags, stream_id, SWITCH_MEDIA_TYPE_VIDEO)) {
                    return SWITCH_STATUS_SUCCESS;
            }
```

为节省篇幅，对上述命令的输出进行了删减，读者可以自己试一下，或者直接访问 Web 版的界面查看：http://fisheye.freeswitch.org/changelog/freeswitch.git/?cs=c4e7c30 。

当然，我们写这个例子的目的是告诉大家，即使是 FreeSWITCH 的作者，也可能会犯错误，所以在汇报 Bug 时不要有过多顾虑。

22.1.5 汇报严重的问题

在上一节，我们鼓励大家要勇敢汇报 Bug。许多朋友可能在遇到问题时不确定哪个问题是 Bug，便在邮件列表中询问，一来二去，耽误了好长时间。FreeSWITCH 的作者 Anthony Minessale 经常在邮件列表中说：“在 Jira 上告诉一个人这不是一个 Bug 比在邮件列表中跟踪这些问题要容易得多”。意思是说，如果感觉类似 Bug 的问题尽管往 Jira 上提，而不要使用邮件列表。因为 Jira 是一个 Bug 跟踪系统，即使你的判断不对，别人也会直接在 Jira 系统上告诉你，流程很清晰。而如果使用邮件列表，因为群里的很多人每天都会收到几百封的邮件，你的信息很容易就被埋没了。

当然，无论如何，如果遇到系统崩溃，那一定是个重大问题。FreeSWITCH 的目标是让它不崩溃，因此所有的崩溃都应该向 Jira 汇报。

笔者就曾经汇报过一个在使用会议系统时系统崩溃的案例。当发现系统崩溃后，笔者根

据崩溃后产生的 core Dump 文件（内核转储文件），发现了一些导致问题的可能原因。下面是当时内核文件的反向跟踪信息（Back Trace，在 GDB 中使用 bt 命令得到）：

```
    Thread 38 (core thread 37):
    #0  switch_core_session_get_channel (session=0x0) at switch_core_session.c:1190
    #1  0x00000001033d5e09 in conference_video_thread_run (thread=0x0,
obj=0x7fd51c8a1f68) at mod_conference.c:1436
    #2  0x00007fff887bc8bf in _pthread_start ()
    #3  0x00007fff887bfb75 in thread_start ()
```

可以看出，在第“#0”个函数调用中，session=0x0，即 session 指针为空指针，导致后续的操作出错。而该函数是在第“#1”个函数调用的时候出现的，它发生在 mod_conference:1436，因此我们很容易在源代码目录中找到它，具体如下：

```
    1436 switch_channel_t *ichannel =switch_core_session_get_channel(imember->session);
```

分析问题我们可以得出，问题的原因在于并不是所有的会议成员（imember）都会对应一个 Channel（如录、放音等虚拟成员）。因此笔者简单生成了一个补丁，提交到 Jira 上。后来 Anthony 在合并的时候又修改了一些内容，因此产生了如下的补丁：

```
        for (imember = conference->members; imember; imember = imember->next) {
    -           switch_channel_t *ichannel =
    -               switch_core_session_get_channel(imember->session);
    +           switch_core_session_t *isession = imember->session;
    +           switch_channel_t *ichannel;
    +
    +           if (!isession || !switch_core_session_read_lock(ises
    +                   continue;
    +           }
    +
    +           ichannel = switch_core_session_get_channel(imember->ses
    ...
    +           switch_core_session_rwunlock(isession);
```

经过测试发现，该补丁虽然解决了崩溃问题，但可能会造成死锁。再次反馈后，发现是又忘了判断 SWITCH_STATUS_SUCCESS 了。通过如下补丁把问题修复。

```
    -    if (!isession || !switch_core_session_read_lock(isession)) {
    +    if (!isession || switch_core_session_read_lock(isession) != SWITCH_
STATUS_SUCCESS) {
```

本案例记录在 http://jira.freeswitch.org/browse/FS-4318 处，感兴趣的读者可以深入研究。

22.1.6 给 Sofia-SIP 打补丁

前面讲的一些例子都是在 FreeSWITCH 代码中打补丁。我们再来看一个更深层次的例子。

为了避免重复发明轮子，FreeSWITCH 使用了大量第三方的代码库。同时为了编译方便，以及减少由于不同版本的引起的可能的混乱，FreeSWITCH 尽量将一些协议兼容的第三方代码库也放到 FreeSWITCH 源代码库中。这些库一般放到 FreeSWITCH 源代码的 libs 目录中。Sofia-SIP 即是其中之一。

笔者以前做过一个项目，需要在 FreeSWITCH 中增加 MSRP⊖协议的支持。而经过跟踪发现，Sofia-SIP 底层根本不支持该协议，因而如果使用该协议，就需要修改 Sofia-SIP 底层的代码。但 Sofia-SIP 库近几年的维护几乎停滞。经过与 FreeSWITCH 官方的沟通，他们说我们可以先把代码提交到 FreeSWITCH 中，等到需要的时候再提交到上游的 Sofia-SIP 库中。

后来，笔者便在 Jira 上提交了一个补丁（http://jira.freeswitch.org/browse/FS-3748）。下面我们来简单看一下这个补丁的内容。

首先，在 Sofia-SIP 库中提交补丁时，需要更新 .update 文件。该文件的内容不重要，一般就是写上当前的日期。通过修改该文件，当再次执行 `make mod_sofia 进行编译时，它便会感知到 Sofia-SIP 库的变化，进而会重新编译 Sofia-SIP 库⊜。

```
diff --git a/libs/sofia-sip/.update b/libs/sofia-sip/.update
@@ -1 +1 @@
-Tue Nov 22 18:16:53 CST 2011
+Tue Dec  6 18:12:20 CST 2011
```

实际的支持代码需要在多个文件中添加，如首先在 sdp_parse.c 中，让它在解析的时候认识 SDP 中 MSRP 相关的内容（否则协议栈会拒绝）：

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sdp_parse.c ...

+  else if (su_casematch(s, "TCP/MSRP"))
+    m->m_proto = sdp_proto_msrp, m->m_proto_name = "TCP/MSRP";
+  else if (su_casematch(s, "TCP/TLS/MSRP"))
+    m->m_proto = sdp_proto_msrps, m->m_proto_name = "TCP/TLS/MSRP";
   else if (su_casematch(s, "UDP"))
     m->m_proto = sdp_proto_udp, m->m_proto_name = "UDP";
   else if (su_casematch(s, "TCP"))
```

在生成 SDP 的时候也要加入 MSRP 支持：

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sdp_print.c ...

     case sdp_proto_rtp:    proto = "RTP/AVP"; break;
     case sdp_proto_srtp:   proto = "RTP/SAVP"; break;
     case sdp_proto_udptl: proto = "udptl"; break;
+    case sdp_proto_msrp:   proto = "TCP/MSRP"; break;
+    case sdp_proto_msrps:   proto = "TCP/TLS/MSRP"; break;
```

最后，我们也把 MSRP 协议相关的常量加到 sdp_proto_e 枚举类型中。这样底层的协议栈就能适当解析出 MSRP 协议的相关内容，剩下的，我们只需要在上层的 mod_sofia 模块中增加相关的支持代码就行了。

⊖ MSRP（Message Session Relay Protocol）称为中继会话中继协定，可用于基于 Session 的即时消息传递或文件传递等，参见 http://tools.ietf.org/html/rfc4975。

⊜ 当然，实际的 Makefile 编译机制应该能自动探测到所有依赖的文件的变化，而不需要这种手工修改自定义的文件。但由于 Sofia-SIP 是第三方的库，为了避免过度耦合，因而采用了这种比较简单的方法。FreeSWITCH 中使用的其他的第三方库也有类似的机制。

```
diff --git a/libs/sofia-sip/libsofia-sip-ua/sdp/sofia-sip/sdp.h

@@ -243,6 +243,8 @@ typedef enum
   sdp_proto_rtp = 256,                /**< RTP/AVP */
   sdp_proto_srtp = 257,               /**< RTP/SAVP  */
   sdp_proto_udptl = 258,              /**< UDPTL. @NEW_1_12_4. */
+  sdp_proto_msrp =  259,              /**< TCP/MSRP @NEW_MSRP*/
+  sdp_proto_msrps = 260,              /**< TCP/TLS/MSRP @NEW_MSRP*/
   sdp_proto_tls = 511,                /**< TLS over TCP */
   sdp_proto_any = 512                 /**< * wildcard */
 } sdp_proto_e;
```

进行上述修改后，Sofia-SIP 库就具备支持 MSRP 协议的能力了。当然，具体的 MSRP 协议支持和处理还需要上层代码（如在 mod_sofia 中增加相应逻辑）的支持，在此我们就不多讲了。在本例中，我们介绍了底层的 Sofia-SIP 库的修改方法以及注意事项。补丁内容的本身并不重要，重要的是了解这里的方法和流程，以便在以后遇到类似问题时进行更深入的研究。在后面，FreeSWITCH 开发者还修复了一些 Bug 并增加了 SIP over WebSocket 支持，以支持 WebRTC。有兴趣的读者也可以看一下这部分的更新历史。

22.1.7 给现有 App 增加新功能

在笔者的某一个咨询项目中，有个客户提到，他想与现有的 WebServer 集成，但又不想使用（比方说 ESL 等）比较复杂的解决方案，问笔者有没有更好的解决方案。笔者推荐他可以直接在 Dialplan 或 Lua 脚本中调用 curl App 跟远程的 HTTP 服务器交互。curl App 是在 mod_curl 中实现的，在 Dialplan 中的使用方法如下：

```
<action application="curl" data="http://..."/>
```

这个方案最初使用起来效果不错，直到某一天遇到个问题——有些 curl 调用由于服务器的卡死导致 Channel 老是卡死，电话挂掉以后还有残留的僵尸数据。出现该问题的原因是 curl 调用远程服务器一直没有返回，因而线程阻塞。经过对源代码进行研究，发现 mod_curl 是使用 libcurl 实现的，但并没有使用超时机制，所以导致了上述问题。

使用开源项目的好处就是我们不仅能修复 Bug，还能随时添加新功能。通过查阅 libcurl 的文档，我们发现可以通过 CURLOPT_CONNECTTIMEOUT 和 CURLOPT_TIMEOUT 选项控制请求超时。其中，前者是连接超时，即多长时间连接不到服务器即为超时；后者是执行超时，即在长时间服务器不返回结果即为超时。

我们发现在 mod_curl 中连接远程服务器并获取文件的函数是在 do_lookup_url 函数中实现的，因此我们很快实现了如下的补丁（有时候，做好事是不需要留名的，但有时候在开源项目中留下自己的名字也感觉挺不错的，因而笔者在该模块前两位贡献者之后留下了自己的名字）：

```
   * Rupa Schomaker <rupa@rupa.com>
   * Yossi Neiman <mishehu@freeswitch.org>
+  * Seven Du <dujinfang@gmail.com>
```

为了存放我们的超时参数，我们定义了一个结构体，并使用 typedef 定义了一个新的类

型 curl_options_t：

```
+struct curl_options_obj {
+       long connect_timeout;
+       long timeout;
+};
+typedef struct curl_options_obj curl_options_t;
```

然后，我们在原来的 do_lookup_url 函数的基础上增加了一个 curl_options_t 指针类型的参数 options：

```
-static http_data_t *do_lookup_url(...,
+static http_data_t *do_lookup_url(..., curl_options_t *options)
```

然后增加如下补丁并进行判断，如果 options 参数存在（保证向后兼容，如果没有提供超时参数，则继续保持原来的行为），则调用 libcurl 的 switch_curl_easy_setopt 函数设置相应的超时参数。

```
+       if (options) {
+           if (options->connect_timeout) {
+               switch_curl_easy_setopt(curl_handle,
+                   CURLOPT_CONNECTTIMEOUT, options->connect_timeout);
+           }
+
+           if (options->timeout) {
+               switch_curl_easy_setopt(curl_handle,
+                   CURLOPT_TIMEOUT, options->timeout);
+           }
+       }
```

至此，do_lookup_url 函数就已经具备超时功能了。但为了使用它，在 curl App 对应的函数中我们需要先初始化一个 optioins 结构体，并从当前的通道变量中收集相关的超时参数：

```
+   curl_options_t options = { 0 };
+   const char *curl_timeout;


+   curl_timeout = switch_channel_get_variable(channel, "curl_connect_timeout");
+   if (curl_timeout) options.connect_timeout = atoi(curl_timeout);
+
+   curl_timeout = switch_channel_get_variable(channel, "curl_timeout");
+   if (curl_timeout) options.timeout = atoi(curl_timeout);
```

初始化完了 options 参数，我们就可以在原来调用的位置把该参数加上了：

```
-   http_data = do_lookup_url(pool, url, method, postdata, content_type);
+   http_data = do_lookup_url(pool, url, method, postdata, content_type, &options);
```

至此，我们增加的功能就应该完成了。不过，后来我们在编译时发现还有一个错误。由于 mod_curl 模块还同时实现了一个 curl API 命令，它也调用了 do_lookup_url 函数。而由于我们修改了 do_lookup_url 函数的定义，故导致无法编译。我们暂时不准备也为该 API 命令增加类似的功能（我们没有用到它），因此为了简单起见，我们仅仅给该调用增加了一个空指针作为参数。

```
-    http_data = do_lookup_url(pool, url, method, postdata, content_type);
+    http_data = do_lookup_url(pool, url, method, postdata, content_type, NULL);
```

编译顺利通过。通过在 Dialplan 中增加如下的设置，我们的问题也顺利解决了。

```
<action application="set" data="curl_connect_timeout=3000"/>
<action application="set" data="curl_timeout=5000"/>
<action application="curl" data="http://..."/>
```

综上，该补丁的实现思路是比较典型的，其遵循了 FreeSWITCH 的架构和设计思想——通过通道变量改变 App 的行为。所以，我们增加了两个通道变量（curl_connect_timeout 和 curl_timeout），并在 curl App 执行过程中根据这两个变量的值决定是否启动超时机制，以及控制合理的超时时间。

该补丁的提交哈希是 d8a02dc，读者可以通过“git show d8a02dc”命令查看。另外，Anthony 后来也在 curl API 命令中增加了该超时特性的支持，详见提交 6ef3f7bd。

22.1.8 给 FreeSWITCH 增加一个新的 Interface

故事要从若干年前说起。当年，笔者学习 FreeSWITCH 时间不长。在测试中文模块（mod_say_zh）时，发现它说出来的中文不符合中文用户的习惯。比如，在用英语读美元时，$10.20 的习惯读法是“10 dollar 20 cents”，而在中文模块中，就顺便读成了“十元二十分”，这显然不符合中文习惯。

当时，笔者就进行了一些改进，并提交了一个补丁，见 http://jira.freeswitch.org/browse/FS-2809。当时笔者做得比较激进，连厘都写上去了，如 10.1234 元将读成“十元一角二分三厘四”。当然，当时只是为了好玩，因为笔者相信当时 FreeSWITCH 圈里的人，应该没有人能比笔者更懂中文了。

不过，在补丁提交了之后，原来模块的维护者 Steve Underwood 却说，笔者做的修改太“中国化”了，如果那样改了，势必不符合世界其他地区的习惯。在此之后笔者才意识到，原来中文是全世界的，而我确实是见识短浅。比方说，如果在美国，即使使用中文读，也确实应该是“十元二十分”啊！也许正是从那以后，笔者考虑问题都会把眼光放远一些了。

但无论如何，一定要是支持中国的中文才叫中文。所以笔者与原作者探讨，是否增加一个通道变量检查之类的（类似于 22.1.8 节我们提到的用通道变量控制相关行为），让用户可以酌情选择？不过一直没有得到回应。后来，Mike Jerris（FreeSWITCH 三剑客之一）提议，可以做一个新的 Interface 时，我才恍然大悟——是啊，怎么没想到这一点？

后来笔者修改了补丁，增加了一个新的 Say Interface——zh_CN。

其实，增加一个 Say Interface 很简单，只需要在 mod_say_zh 中（注意，我们是在该模块中新增加了一个 Interface，而没有增加新的模块）增加如下的接口定义：

```
+        say_zh_CN_interface = switch_loadable_module_create_interface(
             *module_interface, SWITCH_SAY_INTERFACE);
+        say_zh_CN_interface->interface_name = "zh_CN";
+        say_zh_CN_interface->say_function = zh_CN_say;
```

然后，实现 zh_CN_say 回调函数，该函数基本上与原来的 zh_say 函数一样（还是调用跟以前一样的函数），只是在读货币（SST_CURRENCY）的时候，使用了笔者专门实现的只针对中国的 zh_CN_say_money 函数。

```
+static switch_status_t zh_CN_say(...
+{

+        case SST_CURRENCY:
+                say_cb = zh_CN_say_money;
```

关于 zh_CN_say_money 函数具体的算法在此就不多讲了，有兴趣的读者可以参考 Jira 上的链接。

最后，我们再补充点小知识。当时 FreeSWITCH 出现了两个大的分支，一个是 master，一个是 v1.2.stabe。前者是最新的开发版，并将成为新的 1.4 版，而后者将保持 1.2 版的向后兼容。因此，在这个时期，提交代码时要同时提交到两个分支中。

首先，笔者在本地提交了代码，在提交的 Message 信息中注明了“FS-2809 --resolved”，当该提交推到远程的 Git 代码库时，代码库中的钩子程序（hook）会自动与“FS-2809”那条 Jira 报告相关联，并将 Jira 报告的状态设为“Resolved”。

```
$ git ci -m 'FS-2809 --resolved' .
[master 51d3282] FS-2809 --resolved
 1 file changed, 97 insertions(+), 1 deletion(-)
```

然后，通过 cherry-pick 将本次修改合并到 v1.2.stable 分支中：

```
$ git checkout v1.2.stable
Switched to branch 'v1.2.stable'
$ git cherry-pick 51d3282
[v1.2.stable f90e828] FS-2809 --resolved
 1 file changed, 97 insertions(+), 1 deletion(-)
```

最后将本地两个分支的修改推到远程 Git 仓库，让世界了解中国。

```
$ git push ssh master
$ git push ssh v1.2.stable
```

从本例中可以看出，实现一个新的 Interface 也不是很复杂的事。当然，仔细阅读源代码，保持与其他开发者沟通是很重要的。

22.2 写一个新的 FreeSWITCH 编解码模块

我们前面的例子都是在为现有的代码上打补丁，在本节我们看一下如何增加一个新的模块。

在笔者测试 VP8 视频编码时，FreeSWITCH 还不支持 VP8，因而需要自己添加支持。好在，在 FreeSWITCH 中写一个新模块很简单。而且，由于 FreeSWITCH 中的视频模块不支持转码，因而大部分回调函数什么也不做。

我们首先在 FreeSWITCH 源代码目录中 src/mod/codecs 下创建 mod_vp8 目录，并在

里面创建 mod_vp8.c 文件，然后，找一个类似的编解码模块，并把它里面的内容复制过来稍加修改即可。当时笔者发现与 VP8 最像的模块是 mod_theora，因此就直接复制了 mod_threora.c 里面的内容。修改后的 mod_vp8.c 内容如下。

首先，是 include 和模块声明。在该模块中，我们只需要 load 函数。

```
33  #include <switch.h>
34
35  SWITCH_MODULE_LOAD_FUNCTION(mod_vp8_load);
36  SWITCH_MODULE_DEFINITION(mod_vp8, mod_vp8_load, NULL, NULL);
```

在编解码模块中，当在核心中初始化一个编码时，首先回调的就是 init，即这里的 switch_vp8_init 函数。该函数在此要做的事情不多，基本上直接返回了成功值——SWITCH_STATUS_SUCCESS。

```
38   static switch_status_t switch_vp8_init(switch_codec_t *codec, switch_
codec_flag_t flags, const switch_codec_settings_t *codec_settings)
39  {
...
51          return SWITCH_STATUS_SUCCESS;
```

如果在调用该模块进行编码时或解码时，将调用这里的 encode 或 decode 函数。由于我们并不支持视频的编、解码，因此直接返回 SWITCH_STATUS_FALSE。实际上，由于核心本身不支持编解码，因而永远也不会回调到这里。

```
55  static switch_status_t switch_vp8_encode(switch_codec_t *codec,
...
61  {
62      return SWITCH_STATUS_FALSE;
63  }
64
65  static switch_status_t switch_vp8_decode(switch_codec_t *codec,
...
71  {
72      return SWITCH_STATUS_FALSE;
73  }
```

当然，最后要释放编解码器的回调函数 destroy：

```
75  static switch_status_t switch_vp8_destroy(switch_codec_t *codec)
76  {
77      return SWITCH_STATUS_SUCCESS;
78  }
```

其实该模块最重要的就是 load 函数了。在 load 函数中，第 82 行初始化了一个 codec_interface，它是一个 switch_codec_interface_t 类型的指针，说明我们想要创建一个 Codec Interface。第 84 行就紧接着创建了它。第 85 行，将该 codec_interface 安装到核心中去。

```
80  SWITCH_MODULE_LOAD_FUNCTION(mod_vp8_load)
81  {
82      switch_codec_interface_t *codec_interface;
83      /* connect my internal structure to the blank pointer passed to me */
84      *module_interface = switch_loadable_module_create_module_
```

```
interface(pool, modname);
    85      SWITCH_ADD_CODEC(codec_interface, "VP8 Video (passthru)");
```

第 87 行，在该 codec_interface 上增加了一个实现（Implementation），并增加了实现的回调函数。具体的参数定义我们在此就不多讲了，总之它定义了 4 个回调函数，即我们上面讲过的 init、encode、decode 和 destroy（参见 20.3.13 节的内容）。虽然有些回调函数什么也不做，不过我们也最好写上它们，以便跟现有的其他模块中的代码一致。最后，返回 SWITCH_STATUS_SUCCESS 以标明模块加载成功（第 102 行）。

```
    87      switch_core_codec_add_implementation(pool, codec_interface,
    88          SWITCH_CODEC_TYPE_VIDEO, 99, "VP8", NULL, 90000, 90000, 0,
    89          0, 0, 0, 0, 1, 1, switch_vp8_init, switch_vp8_encode,
                                  switch_vp8_decode, switch_vp8_destroy);
    ...
    102     return SWITCH_STATUS_SUCCESS;
```

后来，Anthony 在做 WebRTC 时，还用同样的方法增加了 red 和 ulpfec 视频编码，不过那就是后话了（见第 91 行和第 96 行）。

```
    91      SWITCH_ADD_CODEC(codec_interface, "red Video (passthru)");
    96      SWITCH_ADD_CODEC(codec_interface, "ulpfec Video (passthru)");
```

有了上述的目录和模块实现文件后，在核心进行 configure 的时候将会自动生成一个 Makefile，不过，我们也可以先自己写一个 Makefile 用于测试。在 Makefile 中加入如下内容后就可以编译该模块了。其中第 1 行指定 FreeSWITCH 源代码的主目录，第 2 行装入通用的模块编译规则：

```
BASE=../../../..
include $(BASE)/build/modmake.rules
```

然后，直接在当前目录下可以使用如下命令编译安装：

```
# make install
```

接下来就可以在 FreeSWITCH 中直接加载使用了。当然，最后笔者把该模块也提交给官方了，相关 Jira 的地址是：http://jira.freeswitch.org/browse/FS-4092。

22.3　从头开始写一个模块

在 22.2 节，我们给大家讲了编码、解码模块的实现方法。本节我们再来从头实现一个综合性模块，实现自己的 Dialplan、自己的 App 以及自己的 API。

22.3.1　初始准备工作

在准备下一步之前，我们先要为我们的模块取一个名字。笔者写书写到这里，脑子几乎用尽了，实在想不出更有创意的名字了，不如就叫 mod_book 吧。

我们的 mod_book 也将脱离 FreeSWITCH 源代码的环境，单独存放。因此，你可以在任何喜欢的目录下创建目录 mod_book，然后在里面创建 mod_book.c。内容我们还是参照上

面讲的 mod_vp8.c（它已经足够简单了），将不需要的函数的功能删除后，并把所有的 mod_vp8 替换为 mod_book，得到的新的模块文件如下：

```
01  /* Book Example: Dialplan/App/API  Author: Seven Du */
02
03  #include <switch.h>
04
05  SWITCH_MODULE_LOAD_FUNCTION(mod_book_load);
06  SWITCH_MODULE_DEFINITION(mod_book, mod_book_load, NULL, NULL);
07
08  SWITCH_MODULE_LOAD_FUNCTION(mod_book_load)
09  {
10      *module_interface = switch_loadable_module_create_module_interface(pool, modname);
11      return SWITCH_STATUS_SUCCESS;
12  }
```

可以看出，该模块应该是最简单了，它只有短短的 12 行（去掉注释只有 10 行）。我们迅速创建一个 Makefile，内容如下（注意，我们这里的 BASE 变量引用的是一个绝对路径，它就是 FreeSWITCH 源代码的路径，如果你的源代码路径与笔者的不同，应该相应地修改它）：

```
BASE=/usr/src/freeswitch
include $(BASE)/build/modmake.rules
```

然后，直接在当前目录执行 make install，该模块就安装好了。

接着，到 FreeSWITCH 控制台上加载该模块，从日志输出中可以看到我们的模块已经加载好了：

```
freeswitch> load mod_book
[CONSOLE] switch_loadable_module.c:1464 Successfully Loaded [mod_book]
```

虽然到此为止，我们的模块还什么都不能做，但至少它顺利加载了。我们要把这阶段性的成果记录下来。执行以下 3 条命令将这些成果记录到 Git 中。

```
$ git init
$ git add Makefile mod_book.c
$ git ci -m 'initial commit'
```

22.3.2 写一个简单的 Dialplan

为了使我们的模块更有用，我们需要增加一些功能。在此我们就实现一个自己的 Dialplan Interface——我们仍然叫它 book。下面我们修改 load 函数，首先增加一个变量声明：

```
switch_dialplan_interface_t *dp_interface;
```

然后，在 *module_interface 一行后，向核心注册我们的 Dialplan，并设置一个回调函数：

```
SWITCH_ADD_DIALPLAN(dp_interface, "book", book_dialplan_hunt);
```

然后实现该回调函数。注意这里我们文件中的行号发生了变化。该回调函数是使用 SWITCH_STANDARD_DIALPLAN 声明的。在第 10 行，我们定义了一个 switch_caller_

extension_t 类型的指针变量，用于定义相关的 extension（与 XML Dialplan 中的 <extension> 标签相对应）。第 11 行将得到当前的 Channel。第 14 行将得到一个 caller_profile，它里面保存了主叫用户的相关信息。比如，在第 20 行我们就在日志中打印出了一些我们关心的信息。该信息跟我们最早在第 6 章讲到的“绿色的行”是一样的，**在此我们自己编程，实现了“绿色的行”**。

```
08  SWITCH_STANDARD_DIALPLAN(book_dialplan_hunt)
09  {
10      switch_caller_extension_t *extension = NULL;
11      switch_channel_t *channel = switch_core_session_get_channel(session);
12
13      if (!caller_profile) {
14          caller_profile = switch_channel_get_caller_profile(channel);
15      }
16
17      switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session), SWITCH_LOG_INFO,
18          "Processing %s <%s>->%s in context %s\n",
19          caller_profile->caller_id_name, caller_profile->caller_id_number,
20          caller_profile->destination_number, caller_profile->context);
```

当 FreeSWITCH 执行到该回调函数时，说明有一路电话进入了路由（ROUTING）阶段，我们要查找 Dialplan，返回对应的 Extension（或里面的 Action），以后在后续 Channel 进入执行阶段时（EXECUTE），执行相关的 App。在第 22 行，我们初始化了一个 extension。第 26 行，往该 extension 上增加了一个 App。在此我们并没有进行任何查找，而是直接硬编码了一个 log 的 App。最后，返回我们生成的 extension（第 29 行）。

```
22      extension = switch_caller_extension_new(session, "book", "book");
23
24      if (!extension) abort();
25
26      switch_caller_extension_add_application(session, extension,
27          "log", "INFO Hey, I'm in the book");
28
29      return extension;
30  }
```

进行了这些改变后，我们再次执行 make install，并在 FreeSWITCH 控制台上使用 reload mod_book 重新加载模块。然后可以快速使用如下命令实验一下该 Dialplan 的效果：

```
freeswitch> originate user/1006 9999 book
```

还记得我们在 6.7 节所说的 Dialplan 的三要素吧？其中，9999 就是 Extension，book 就是 Dialplan 的名字；而 Context 由于省略了，默认就是 default。因此，可以在日志中看到如下的“绿色的行”，并且也可以看到我们增加的 App 也如期执行了（输出了对应的日志）。

```
[INFO] mod_book.c:17 Processing  <0000000000>->9999 in context default
[INFO] mod_dptools.c:1595 Hey, I'm in the book
```

至此，我们的 Dialplan 应该可以正常工作了。我们可以在 XML Dialplan 里转向它：

```
<action application="trasfer" data="9999 book default" />
```

也可以在 Sofia Profile 中（如 internal）直接使用它，配置如下：

```
<param name="dialplan" value="book"/>
<param name="context" value="default"/>
```

当然，我们的 Dialplan 功能还不是很强大，有待于进一步加强。我们继续将它提交到 Git 中（在提交之前执行一下 git diff 是个好习惯）：

```
$ git status
$ git diff
$ git commit -m 'add Dialplan Interface' .
```

到这里我们应该更深入地理解 Dialplan 到底是干什么的——**它就是负责找到一组 App，保证以后 FreeSWITCH 后续能执行这些 App。**

22.3.3 增加一个 App

在上述的 Dialplan 的例子中，我们还是使用了现有的 log App 作为例子。下面我们该实现一个自己的 App 了。我们继续将该 App 取名为 book。实现的步骤如下。

首先，声明一个 app_interface：

```
switch_application_interface_t *app_interface;
```

将该 App 向核心注册，并增加一个回调函数 book_function：

```
SWITCH_ADD_APP(app_interface, "book", "book example", "book example",
                              book_function, "[name]", SAF_SUPPORT_NOMEDIA);
```

实现该回调函数。该函数的参数将从 data 指针中传过来，如果为空（第 39 行），则我们给它指定一个默认的名字；否则，就把传入的参数作为书的名字（第 42 行）。第 45 行输出一条日志，打印自己的名字：

```
35  SWITCH_STANDARD_APP(book_function)
36  {
37      const char *name;
38
39      if (zstr(data)) {
40          name = "No Name";
41      } else {
42          name = data;
43      }
44
45      switch_log_printf(SWITCH_CHANNEL_SESSION_LOG(session),
46          SWITCH_LOG_INFO, "I'm a book, My name is: %s\n", name);
47  }
```

当然，我们也不忘了在我们刚才的 Dialplan 中的 extension 上加上我们自己实现的 App：

```
switch_caller_extension_add_application(session, extension,
    "book", "FreeSWITCH - The Definitive Guide");
```

重新编译加载后，再执行一次，日志如下（可以看出，我们的 App 已经执行了）：

```
[INFO] mod_dptools.c:1595 Hey, I'm in the book
```

```
[INFO] mod_book.c:45 I'm a book, My name is: FreeSWITCH - The Definitive Guide
```

最后，当然我们也不要忘了再把我们的改变提交到 Git 里：

```
$ git commit -m 'add book App' .
```

22.3.4 写一个 API

通过上面的例子，相信读者也能想到，自己写一个 API 也是很容易的。出于完整性考虑，我们就来再写一个。我们已经完全没有创意了，因此该 API 的名字还是叫 book。

声明一个 api_interface：

```
switch_api_interface_t *api_interface;
```

将 API 注册到核心，并设置回调函数 book_api_function：

```
SWITCH_ADD_API(api_interface, "book", "book example", book_api_function, "[name]");
```

实现回调函数。该回调函数与上面的 book_function 类似，不同的是，参数是从 cmd（第 53 行）获取的，而且这里我们没有打印日志，而是直接将命令的返回结果写到输出流（stream）里去了（第 59 行）。

```
49  SWITCH_STANDARD_API(book_api_function)
50  {
51      const char *name;
52
53      if (zstr(cmd)) {
54          name = "No Name";
55      } else {
56          name = cmd;
57      }
58
59      stream->write_function(stream, "I'm a book, My name is: %s\n", name);
60
61      return SWITCH_STATUS_SUCCESS;
62  }
```

重新编译并加载该模块后，我们在日志中看到如下的输出，它分别增加了以 book 为名称的 Dialplan、Application 和 API Function（我们在前面还没注意到呢）。

```
[NOTICE] switch_loadable_module.c:227 Adding Dialplan 'book'
[NOTICE] switch_loadable_module.c:269 Adding Application 'book'
[NOTICE] switch_loadable_module.c:315 Adding API Function 'book'
```

然后，我们在 FreeSWITCH 控制台上就可以执行 book 命令了：

```
freeswitch> book
I'm a book, My name is: No Name

freeswitch> book FreeSWITCH - The Definitive Guide
I'm a book, My name is: FreeSWITCH - The Definitive Guide
```

再一次，我们将里程碑成果提交到 Git 中：

```
$ git commit -m 'add book API Interface' .
```

22.3.5 小结

在本节的例子中，我们从无到有一步一步实现了一个模块，添加了 Dialplan、App 以及 API。在前面的例子中，虽然我们阅读了很多源代码，但是“纸上得来终觉浅”，唯有手工一步一步地做一次才知到每一步是怎么来的。

另外，我们也把每一步的结果都提交到 Git 版本管理系统中了，因而可以随时查阅历史，重温这段美好的回忆。各位读者可以自己试一下，也可以从本书的 Github 站点上翻阅在写作本书时的美好回忆。

22.4 使用 libfreeswitch

FreeSWITCH 不仅有完善的可加载模块支持，而且它的库 libfreeswitch 也可以被连接到其他系统中去，使得其他系统立即具有所有的 FreeSWITCH 中的功能。

22.4.1 自己写一个软交换机

下面我们就尝试自己写一个软交换机。这里说的自己写，并不是一切都从头写，而是利用现有的 libfreeswitch 库，把它集成到我们的系统中来。

假设我们已有了一个系统，该系统的功能非常强大。不过，为了便于讲解，我们把系统精简到了最简单的程序，即精简到它在执行后仅仅打印一条信息就退出。

```
int main(int argc, char **argv)
{
    printf("Hello, MySWITCH is running ...\n");
    return 0;
}
```

下面我们要将 libfreeswitch 集成进我们的系统中，因此我们对 main 函数做了一些改变，具体实现代码如下。其中，我们在第 3 行装入了 switch.h 头文件，以便我们能引用里面的函数；第 7 行，我们设置一个 flags 标志，让它使用核心数据库；第 8 行，定义一个 console 变量并设为 true；第 13 行，设置一些默认的全局参数；第 14 行，初始化并加载模块；第 15 行，进入控制台循环。

```
01  /* MySwitch using libfreeswitch */
02
03  #include <switch.h>
04
05  int main(int argc, char** argv)
06  {
07      switch_core_flag_t flags = SCF_USE_SQL;
08      switch_bool_t console = SWITCH_TRUE;
09      const char *err = NULL;
10
11      printf("Hello, MySWITCH is running ...\n");
12
13      switch_core_set_globals();
```

```
    switch_core_init_and_modload(flags, console, &err);
    switch_core_runtime_loop(!console);
    return 0;
}
```

通过这短短的几行，我们就写了一个功能强大的交换机，它具有 FreeSWITCH 全部的功能。我们通过如下的 Makefile 来编译它：

```
FS  = /usr/local/freeswitch
INC = -I$(FS)/include
LIB = -L$(FS)/lib

all: myswitch

myswitch: myswitch.c
    gcc -o myswitch -ggdb $(INC) $(LIB) -lfreeswitch myswitch.c
```

在上面的 Makefile 中，最开始的 3 行我们定义了 3 个变量。其中 INC 和 LIB 分别指定头文件和库文件的参数。最后一行使用 gcc 进行编译，输出可执行文件为 myswitch；为了调试方便，我们在编译时使用 -ggdb 加入符号表；-lfreeswitch 作用为连接 libfreeswitch.so 库文件（在 Mac 上为“freeswitch.dylib”），最后的 myswitch.c 为源文件名。

执行 make 即可进行编译，编译完成后运行结果如下。由结果可以看到其为与 FreeSWITCH 中类似的日志，由此证明，一个强大的软交换机诞生了。

```
./myswitch
Hello, MySWITCH is running ...
2013-12-10 20:50:52.349175 [INFO] switch_event.c:669 Activate Eventing Engine.
...
```

22.4.2　使用 libfreeswitch 提供的库函数

在大多数情况下，我们不会重新发明一个 FreeSWITCH，而是想使用它提供的库文件中有用的部分。在下面这个例子中，我们就用到了它的文件接口、编码转换接口及 RTP 等功能。

我们程序的功能是从本地音频文件中读取数据，然后用 PCMU 进行编码，并通过 RTP 发送出去。将源文件命名为 myrtp.c。在 main 函数的最开始，我们定义并初始化了很多变量。在此我们先不介绍这些变量，等后面用到的时候有必要的话再讲。该文件内容如下：

```
/* File/Codec/RTP Example Author: Seven Du */

#include <switch.h>

int main(int argc, char *argv[])
{
    switch_bool_t verbose = SWITCH_TRUE;
    const char *err = NULL;
    const char *fmtp = "";
    int ptime = 20;
    const char *input = NULL;
    int channels = 1;
```

```
13      int rate = 8000;
14      switch_file_handle_t fh_input = { 0 };
15      switch_codec_t codec = { 0 };
16      char buf[2048];
17      switch_size_t len = sizeof(buf)/2;
18      switch_memory_pool_t *pool = NULL;
19      int blocksize;
20      switch_rtp_flag_t rtp_flags[SWITCH_RTP_FLAG_INVALID] = { 0 };
21      switch_frame_t read_frame = { 0 };
22      switch_frame_t write_frame = { 0 };
23      switch_rtp_t *rtp_session = NULL;
24      char *local_addr = "127.0.0.1";
25      char *remote_addr = "127.0.0.1";
26      switch_port_t local_port = 4444;
27      switch_port_t remote_port = 6666;
28      char *codec_string = "PCMU";
29      int payload_type = 0;
30      switch_status_t status;
```

从命令行参数获取音频文件名并将其放到 input 变量中（第 34 行）。

```
32      if (argc < 2) goto usage;
33
34      input = argv[1];
```

第 36 行初始化 libfreeswitch 的内核，这里我们使用了 SCF_MINIMAL 选项，它将启动最小配置（因为这里不需要完整的 FreeSWITCH）。

```
36      if (switch_core_init(SCF_MINIMAL, verbose, &err) != SWITCH_STATUS_SUCCESS) {
37          fprintf(stderr, "Cannot init core [%s]\n", err);
38          goto end;
39      }
```

第 41 行，设置一些全局的参数。第 42 行初始化可加载模块的设置，我们使用了 SWITCH_FALSE 参数不让它自动加载模块，而是在后面手工加载。比如，第 43 ~ 44 行就加载了两个核心的模块，它们都是在核心中实现的，CORE_SOFTTIMER_MODULE 是一个时钟模块，用于定时；CORE_PCM_MODULE 即 PCM 编解码模块，用于 PCMU/PCMA 编解码。由于我们这里只用到 PCMU，因此，其他编解码模块就不需要加载了，否则需要手工加载对应的编解码模块。由于我们要读取音频文件，因此我们在第 47 行加载了 mod_sndfile 模块，它使用 libsndfile 库支持很多类型的声音文件，如 .au、.aiff 等。当然，读者通过前面的学习也可能会想到，如果这里我们需要支持 MP3 的话就需要加载 mod_shout 了。

```
41      switch_core_set_globals();
42      switch_loadable_module_init(SWITCH_FALSE);
43      switch_loadable_module_load_module("", "CORE_SOFTTIMER_MODULE",
            SWITCH_TRUE, &err);
44      switch_loadable_module_load_module("", "CORE_PCM_MODULE", SWITCH_TRUE, &err);
45
46      if (switch_loadable_module_load_module((char *) SWITCH_GLOBAL_dirs.mod_dir,
47          (char *) "mod_sndfile", SWITCH_TRUE, &err) != SWITCH_STATUS_SUCCESS) {
48          fprintf(stderr, "Cannot init mod_sndfile [%s]\n", err);
49          goto end;
50      }
```

第 53 行初始化一个内存池。第 57 行调用 switch_core_file_open 打开输入的音频文件。其参数的值我们都在 main 函数的开始部分定义了。其中，channels 为声道的数量，rate 为采样率，读者可以倒回去查看一下。如果音频文件中的参数与这里的不匹配，它将按我们这里指定的自动进行转换。

```
53      switch_core_new_memory_pool(&pool);
54
55      fprintf(stderr, "Opening file %s\n", input);
56
57      if (switch_core_file_open(&fh_input, input, channels, rate,
58          SWITCH_FILE_FLAG_READ | SWITCH_FILE_DATA_SHORT, NULL) != SWITCH_
              STATUS_SUCCESS) {
59          fprintf(stderr, "Couldn't open %s\n", input);
60          goto end;
61      }
```

第 63 行初始化 PCMU 编解码 codec。音频数据从文件中读出来后，都是以 L16 编码实现的线性编码，后面我们需要把它们转成 PCMU。其中，rate 为采样率、ptime 为打包时间、channels 为声道数。SWITCH_CODEC_FLAG_ENCODE 标志说明我们只需要用到该编码器的编码器，即不需要用它解码。

```
63      if (switch_core_codec_init(&codec,
64          codec_string, fmtp, rate, ptime, channels,
65          SWITCH_CODEC_FLAG_ENCODE, NULL, pool) != SWITCH_STATUS_SUCCESS) {
66          fprintf(stderr, "Couldn't initialize codec for %s@%dh@%di\n",
                codec_string, rate, ptime);
67          goto end;
68      }
```

我们可以根据采样率和打包时间算出一个数据包的长度 len 和需要的内存空间 blocksize（第 78 行），在此我们使用的 PCMU 编码的数据长度就是 $8000 \times 20 / 1000 = 160$，即每个 RTP 包有 160 个字节的数据，而原始读取来的数据由于是使用 16 位的存储，因此每个数据有 2 字节，所以实际原始数据的长度是 160×2 字节 = 320 字节。

```
78      blocksize = len = (rate * ptime) / 1000;
79      switch_assert(sizeof(buf) >= len * 2);
80      fprintf(stderr, "Frame size is %d\n", blocksize);
```

接下来，在第 74 行初始化系统 RTP 环境，然后初始化一个 RTP 的标志参数；第 76 行表示它支持输入输出；第 77 行表示采取非阻塞的方式发送；第 78 行表示允许调试，它将在日志中打印调试信息；第 79 行指定使用时钟，以更好地定时。然后，在第 81 ~ 84 行初始化一个 rtp_session，它的参数包含了 RTP 中必要的参数：本地 IP 地址和端口、远程 IP 地址和端口、负载类型（Payload Type）、采样率以及打包间隔等。另外，soft 是一个定时器的名字，它是核心提供的定时器。

```
74      switch_rtp_init(pool);
75
76      rtp_flags[SWITCH_RTP_FLAG_IO] = 1;
77      rtp_flags[SWITCH_RTP_FLAG_NOBLOCK] = 1;
```

```
78        rtp_flags[SWITCH_RTP_FLAG_DEBUG_RTP_WRITE] = 1;
79        rtp_flags[SWITCH_RTP_FLAG_USE_TIMER] = 1;
80
81        rtp_session = switch_rtp_new(local_addr, local_port,
82            remote_addr, remote_port,
83            payload_type, rate / (1000 / ptime), ptime * 1000,
84            rtp_flags, "soft", &err, pool);
```

libfreeswitch 默认会捕获各种信号，因此，我们在第 99 行将信号捕获回调设为空值，以后我们在调试的时候随时可以按“Ctrl+C”终止程序。

```
91        signal(SIGINT, NULL); /* allow break with Ctrl+C */
```

接下来就是无限循环一直从文件中读取数据。我们每次只读取一帧（len）大小的数据，数据将读到 buf 缓冲区中（第 93 行）。然后，在第 100 ~ 101 行，将读到的数据进行编码，编码后的数据将存储到 encode_buf 中，数据长度可以在 encodec_len 中得到。

```
93        while (switch_core_file_read(&fh_input, buf, &len) ==
94            SWITCH_STATUS_SUCCESS) {
95            char encode_buf[2048];
96            uint32_t encoded_len = sizeof(buf);
97            uint32_t encoded_rate = rate;
98            unsigned int flags = 0;
99
100           if (switch_core_codec_encode(&codec, NULL, buf, len*2, rate,
101               encode_buf, &encoded_len, &encoded_rate, &flags) != SWITCH_STATUS_SUCCESS) {
102               fprintf(stderr, "Codec encoder error\n");
103               goto end;
104           }
```

将数据编码成 PCMU 以后，我们就可以把它打包成一个数据帧（frame）。下面就是设置该数据帧的各种参数：第 107 行，设置帧数据的地址指向我们新编码的数据；第 108 行设置数据的长度；第 109 行设置缓冲区的长度；第 110 行设置采样率；第 111 行设置该数据帧的编解码；然后在第 112 行将该数据帧发送出去。

```
106            len = encoded_len;
107            write_frame.data = encode_buf;
108            write_frame.datalen = len;
109            write_frame.buflen = len;
110            write_frame.rate= 8000;
111            write_frame.codec = &codec;
112            switch_rtp_write_frame(rtp_session, &write_frame);
```

第 114 行，我们尝试从该 rtp_session 中读取一帧数据。其实，由于没人给我们发送数据，它将于 20 毫秒后超时，进入下一次循环。当然，在进入下一次循环前我们要重置 len 的值（第 121 行），以避免 len 的值可能在某些场合下更改为其他的值从而引起的错误。

```
114            status = switch_rtp_zerocopy_read_frame(rtp_session,
                           &read_frame, 0);
115
116            if (status != SWITCH_STATUS_SUCCESS &&
117                status != SWITCH_STATUS_BREAK) {
118                goto end;
```

```
119         }
120
121             len = blocksize;
122     }
```

如果在前面遇到错误，或前面读文件的循环退出（如，读到文件尾），则代码会执行到第 124 行。后面第 125 行会释放编解码器；第 126 行关掉文件接口；第 127 行释放内存池；并于第 128 行释放整个 libfreeswitch 的核心资源，程序结束。

```
124 end:
125     switch_core_codec_destroy(&codec);
126     if (fh_input.file_interface) switch_core_file_close(&fh_input);
127     if (pool) switch_core_destroy_memory_pool(&pool);
128     switch_core_destroy();
129     return 0;
```

当然，如果用户在命令行上输入错误的参数，程序将跳到 usage 标签，打印帮助信息。

```
131 usage:
132     printf("Usage: %s input_file\n\n", argv[0]);
133     return 1;
134 }
```

将上述程序编译运行后，便可以看到它从本地的 4444 端口向外（6666 端口）发送 RTP 数据了。由于数据长度为 160 字节，加上 12 字节的 RTP 包头，因而日志中显示的一共是 172 字节。部分日志如下：

```
$ ./myrtp /wav/test.wav
Opening file /wav/test.wav
Frame size is 160
[DEBUG] switch_rtp.c:3047 Starting timer [soft] 160 bytes per 20ms
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=160 m=1
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=320 m=0
W NoName b= 172 127.0.0.1:4444 127.0.0.1:6666 127.0.0.1:4444 pt=0 ts=480 m=0
```

如果在运行时，不想要 FreeSWITCH 打印日志，可以在第 7 行将 verbose 调为 SWITCH_FALSE。读者也可以尝试修改其他参数。

读到这里，也许有的读者会问：数据是否真的发送出去了？怎么验证呢？最简单的答案是，如果你需要这个程序，也许你已经有方法接收了。当然，如果没有的话，也可以把上述程序稍加改造——我们在第 114 行已经有接收 RTP 的代码，只需照着再写一个程序，将收到的数据保存到声音文件中，或者从声卡中放出来。这些就留给读者自行练习了。

22.4.3 其他

其实，FreeSWITCH 的源代码中，也自带一些例子，其中就包括使用 libfreeswitch 的例子。经常有朋友问到——FreeSWITCH 安装目录下的 bin 目录中，除了 freeswitch 和 fs_cli 外，其他的几个程序是干什么用的？其实，要回答这个问题很简单，只需不带参数运行一下那个程序，或者加上“-h”参数，就很容易得到一个帮助信息。如，从 fsxs 程序的帮助信息看，是帮助编译一个模块的。笔者试了一下用它编译 22.3 节的 mod_book，不用 Makefile

也能编译（意味着不用源代码环境也可以编写模块）。比如，下面的命令将生成 mod_book.so

```
$ /usr/local/freeswitch/bin/fsxs build mod_book.so mod_book.c
CC  mod_book.c
LD  mod_book.so [mod_book.o]
```

使用下列命令就可以安装模块了：

```
/usr/local/freeswitch/bin/fsxs install mod_book.so
```

另外，fs_encode 程序可以将一个语音文件从一种编码转到另一种编码，tone2wav 则帮助你从一个 TGML 标记语言描述的铃声转换成一个声音文件。如果要使用这两个程序，读者可以自行参考相关的帮助信息。另外，既然我们已经学会了查看源代码，自然可以在源代码中发现更多的秘密。这两个程序对应的源代码都在 FreeSWITCH 的源代码目录中（fs_encode.c 和 tone2wav.c），跟我们 22.4.2 节的实现方式差不多，也都用到了 libfreeswitch。读者可以找到这几个程序的源代码自己研究一下，并对比一下与我们在本节的实现有何异同。

22.5 调试跟踪

在编写程序时，很少会一次写对，笔者也不例外。在笔者写 myrtp.c 的例子时，就遇到一个 Bug，导致在程序运行时出现 "Segmentation fault"，具体如下：

```
$ ./myrtp /wav/test.wav
Opening file /wav/vacation.wav
Frame size is 160

Segmentation fault: 11 (core dumped)
```

为了查找失败原因，笔者使用 GDB 进行调试。注意，在调试之前，需要先在编译过程中使用 "-ggdb" 选项，以让 GCC 在产生的可执行程序中写入相关的符号表。另外还要注意，设置 ulimit 环境，以允许系统产生内核转储文件（core dump）。比如，笔者使用如下命令设置允许内核转储文件的大小限制（unlimited 即无限制，默认是 0）：

```
ulimit -c unlimited
```

设置完成后，可以使用 "ulimit -a" 命令验证。然后，重新运行程序，产生 core dump 文件。在 Linux 系统上，core dump 文件一般是在当前目录中产生，在笔者使用的 Mac 每上，它固定产生在 /cores 目录中，并以当前的进程号作为扩展名。

在找到的 core dump 文件中，笔者使用以下命令开启了 gdb，装入 core dump 文件用于调试：

```
$ gdb -core /cores/core.75886
GNU gdb 6.3.50-20050815 (Apple version gdb-1824)...
Reading symbols for shared libraries . done
#0  apr_palloc (pool=0x0, size=72) at apr_pools.c:603
603     if (pool->user_mutex) apr_thread_mutex_lock(pool->user_mutex);
```

其中，我们可以看到调用堆栈中的第 "#0" 层有一个 pool 变量是空指针（0x0 即

NULL)，这可能是我们遇到问题的原因。但上述的命令并没有列出详细的调用栈，apr_pools.c 是 APR 底层的库，因而我们还是需要从更上层查找问题。接着输入“bt”命令（Back Trace），我们看到了详细的调用栈，原来在调堆栈的第“#2”层调用 switch_rtp_init 时 pool 指针就是空指针。

```
    (gdb) bt
    #0  apr_palloc (pool=0x0, size=72) at apr_pools.c:603
    #1  0x0000000108db8b55 in apr_thread_mutex_create (mutex=0x108f146d0, flags=1,
pool=0x0) at thread_mutex.c:50
    #2  0x0000000108d53373 in switch_rtp_init (pool=0x0) at switch_rtp.c:1328
    #3  0x0000000108cce7e0 in main (argc=2, argv=0x7fff56f32760) at myrtp.c:74
```

我们查找源文件，发现该 pool 在任何地方都没有初始化，因而它是一个空指针。所以，增加了第 53 行的对内存池初始化的函数后，一切就都正常了。

在本例子中，错误比较明显，因而很容易发现。而在实际编程开发时，可能遇到一些更隐秘的错误，不容易直接从 Back Trace 中看到结果。那就要配合在代码中添加日志打印语句，或临时注释掉一些语句等手段来调试了。有时候，直接使 GDB 连接（attach）到正在运行的进程上，再进行调试、添加断点等，也是比较有效的调试方法。调试程序是一门细活，也需要有一定的耐心和经验。在此我们仅通过此简单的例子，给大家讲解一下调试程序的基本原理和方法，剩下的就需要读者自己多加研究和练习了。

22.6 小结

本章我们主要讲了基于 FreeSWITCH 进行二次开发的知识。直接在 FreeSWITCH 源代码上进行开发，使我们可以直接访问 FreeSWITCH 提供的库函数和 API，对呼叫的控制就更加灵活方便。而且，开发完成后，由于不需要依赖第三方的系统，部署起来也非常简单。

在本章，我们从最简单的汇报 Bug 时的注意事项开始，到真实的汇报 Bug 的例子。其中，我们精心选择了一些实际的案例，每个案例都涵盖一个或多个知识点，并对前面学到的内容进行了复习和补充。

另外，我们也讲了给 FreeSWITCH 增加新特性以及添加新模块的实例，并带领大家从头开始写了一个新模块，添加了我们自己实现的各种 Interface。这使我们从前面章节中的纸上谈兵过渡到了真正的战场上。

然后，我们还讲了如何把 FreeSWITCH 嵌入其他系统的例子，解答了许多朋友经常问到的一些问题。

最后，我们还配合本书的例子讲解了使用 GDB 进行调试的实例，作为开发中的一种辅助手段，相信这也会对广大读者有所帮助。

在所有的实例中，我们都注意穿插讲解了以 Makefile 为主的编译设置，以及以 Git 作为版本控制系统的源代码版本控制，以帮助读者更快更好地将学到的知识应用到实际项目中去。

总之，这些案例虽然由于本书篇幅的限制写得都不长，但都最大限度地涵盖了相关的知识点和应该注意的问题，并给读者提供了深入学习和研究的方向和思路。

诚然，有时候对程序代码的一些解释是繁琐和冗长的，有时候甚至再多的解释也比不过一句代码能说明问题。最正确最好的例子都在 FreeSWITCH 的源代码中。所以，希望读者在本书的基础上，多看源代码，且学且实践。

截至本章，我们就要跟大家说再见了。“师傅领进门，修行在个人”，预祝大家都能很快精通 FreeSWITCH，并能用 FreeSWITCH 开发出最好的应用！

附录 A

FreeSWITCH 开源软件社区指南

FreeSWITCH 是一个开源项目，也是一个开发者社区。FreeSWITCH 的开发很活跃，开发者及社区的成员都很友好。我们欢迎所有人加入 FreeSWITCH 社区，一起学习、讨论，并贡献自己的力量，使我们的社区和我们的项目越办越好。

A.1 中文社区

- http://www.freeswitch.org.cn。FreeSWITCH 中文社区，名称是 FreeSWITCH-CN，社区的最新消息都会发布在这里。
- http://groups.google.com/group/freeswitch-cn。Google Groups 的中文邮件列表。
- 专业的 QQ 群：190435825。
- 微信公众平台：FreeSWITCH-CN。

我们每年都会不定期举办中国区的 FreeSWITCH 沙龙线下活动。这些活动信息将在 www.freeswitch.org.cn 网站上发布。

A.2 英文社区

- http://www.freeswitch.org。FreeSWITCH 的官方网站。
- http://wiki.freeswitch.org。英文的 Wiki 网站，资料丰富。
- http://lists.freeswitch.org。一个邮件列表（大部分用户可以从 freeswitch-users 邮件列表开始）。
- http://freenode.net。一个 IRC 站点，其中有一个供大家交流的“#freeswitch”频道，类似于中国的 QQ 群。
- http://jira.freeswitch.org。一个缺陷跟踪系统，可以在这里汇报 Bug，或者提新需求。如果希望自己的新需求快速得到响应，可以提交一个 Bounty（Bounty 的意思是你希

望实现一个新需求并愿意为之付费)。

- http://files.freeswitch.org。提供各种 FreeSWITCH 资源的下载，包括源代码和已编译的程序等。
- http://cluecon.com。ClueCon [1]的主站。
- http://wiki.freeswitch.org/wiki/Weekly_Conference_Call。列出了 FreeSWITCH 电话会议[2]的参与方式及历史会议的录音。

[1] ClueCon 是一年一度的开发者技术交流会，每年 8 月份在美国芝加哥举行。

[2] 电话会议是 FreeSWITCH 社区成员使用语音交流的一种方式。基本上是每周三 UTC 时间下午 6 点开始，每周五不定时也有 Friday Free For All 的会议。支持使用 WebRTC、Flash、SIP、H323 以及 PSTN 等多种方式参加会议。

附录B

关于 FreeSWITCH 常用术语翻译的意见

FreeSWITCH 的大多数资料都是英文的，因而很多情况下，为了方便阅读或者与其他人交流，都需要将原文翻译成中文。下面就简单谈一下笔者对翻译的理解及针对 FreeSWITCH 的特定术语的翻译。

众所周知，翻译的最高境界是“信、达、雅”，通俗来讲就是“正确、易懂、得体”，它实际上是从“内容、表达、风格”三方面来要求的。但拘泥于以上标准往往会受到很多限制，尤其是对于科技作品来说，用上述观点来评价翻译作品则更难保证评价的正确性和客观性。

一般来说，翻译包括原著与目标语言读者两个对象。那么，我们在翻译时，除了应该忠实于原文外，更应该考虑的是目标读者。比方说，某件事情或动作，对于原著语言的读者来说可能是习以为常的，但对于目标语言的读者来说可能很陌生或根本不知道是怎么回事。这时我们就应该允许译者加入相关背景知识的脚注，或者直接用目标语言读者熟悉的、类似的表达方式来表达。

FreeSWITCH 资料大部分来自其官方 Wiki 或者邮件列表，是来自全球众多网友的贡献。其作者大部分是程序员或技术人员，因此他们大都不太注重语法和修辞，并且这种网络媒体写起来本身就比较自由，再加上还有好多俚语和网络语言，甚至有不少文章也是出自非英语母语的人之手，这就更难保证语法的正确性了，因而翻译起来就非常困难了。在此，笔者主张遇到某些难以翻译的句子时，尽量把长句拆成短句，在充分了解原作的基础上甚至可以用自己的语言表达（Anthony 最喜欢写长句了）。因为我们最重要的任务让读中文的人能理解它，而不是一味地“忠于”原文。

关于英文中的术语，可译可不译。如：没有人会把“HTTP”译成“超文本传输协议”，也没有人把“XML”译成“可扩展标记语言”。为什么？因为它们是众所周知的，至少在技术领域是这样。但对于某些特定领域的术语，有一些甚至在中文中没有对应的词，怎么办呢？最简单的答案是不译，这些既能最大程度地忠于原文，又不会让读者不知所云。当然，如果你的水平很高，你创造一个新词也可能成为国际标准，那就另当别论了。

对于某些缩写，最好能标注原文。有些书会在附录里列一个术语对照表，但笔者认为，

大多数情况下还是直接注在正文（用括号或脚注）中较好，而且只需在第一次出现时加注。如 PSTN（Public Switched Telephone Network，公共电话交换网)、ENUM（E.164 NUmber Mapping，E164 号码映射)。当然，对于非专业的读者来说，他们可能还是不知道是什么，如果你面向的是这样的读者，那最好在脚注里（或超文本里）加个指向比较权威资料（如维基百科）的链接。举个例子，大部分人都不知道自己家电话所在的网络叫“公用电话交换网”，更不用说 PSTN 了。那么，在某些场合说到它与 VoIP 的区别时，我们一般称之为“普通电话网”或“传统电话网”。

以下是本书中笔者对 FreeSWITCH 相关术语的译法，不当之处请广大读者批评指正。

- Dialplan：拨号计划（虽然在 Asterisk 的书中译为“拨号方案”，但笔者还是认为“拨号计划”较好)。
- Endpoint：终点（专有名词，可不译)。
- ASR/TTS：自动语音识别 / 文本语音转换（语音合成)。
- Directory：FreeSWITCH 中的用户目录，或者通用的目录服务（如 LDAP)。
- Events：事件。
- Event handlers：事件句柄。
- Event Socket：事件套接字（可不译)。
- Formats：格式。
- Loggers：日志。
- Languages：语言。
- Say：说（可不译)。
- Phrase：短语（可不译)。
- Timers：定时器，或译为计时器。
- Applications：应用。
- FSAPI：FreeSWITCH 应用程序接口，或译为 FreeSWITCH 命令接口。
- CDR：呼叫详单，或译为呼叫详细记录。
- PSTN：公共电话交换网。
- SIP：会话初始协议。
- SIP UA：SIP 用户代理。
- UAC/UAS：用户代理客户端 / 服务器端。
- Sofia Profile：Sofia 配置文件（可不译)。
- Gateway：网关。
- VoIP：不译。

附录 C
模块列表

FreeSWITCH 的模块有很多，它们丰富了 FreeSWITCH 的功能。实际上，常用的模块本书已经基本都介绍过了。但总有好奇的读者会问：不知道 mod_xxx 这个模块是干什么用的？因此，这里列出大部分已知的模块，供广大读者参考。有的模块在前面介绍得很详细了，在这里就从简了，另外有些模块笔者也没有用过，说错了不如不说，因而也会一笔带过。

关于模块的介绍，在官方的 Wiki 页面上（http://wiki.freeswitch.org/wiki/Modules）大部分都能查到。部分模块也有自己的页面，它们的 URL 也比较规范，如我们已经介绍过的 http://wiki.freeswitch.org/wiki/Mod_sofia、http://wiki.freeswitch.org/wiki/Mod_dptools、http://wiki.freeswitch.org/wiki/Mod_commands 等。

我们基于 FreeSWITCH 1.2.22 版，以自然的顺序来讲解。这里模块的顺序是在 FreeSWITCH 源代码目录中用以下命令生成的：

```
$ cd src/mod
$ find . -type d -name mod_*
```

各个模块根据其主要功能和提供的接口的不同被分配不同的目录中。

1. applications

该目录下的模块提供了大部分应用的功能，有的模块因实现了多种 Interface 而不好归类，这些模块也会存在该目录中。

- mod_abstraction：用于创建新的 API 命令。新的命令可以基于原有的 API 创建，相当于创建原有命令的别名或快捷方式。
- mod_avmd：avmd 是 Advanced Voice Mail Detection 的缩写，即高级语音邮件检测。它是 mod_vmd 的高级版。详见 mod_vmd。
- mod_blacklist：黑名单功能。它提供了一些通过 Dialplan 来添加、删除以及检查黑名单的方法。
- mod_callcenter：一个比较强大的呼叫中心类应用。
- mod_cidlookup：用于主叫号码查询，即可根据主叫号码从本地数据库或网络上

查询到主叫的名字。网络上有开放的服务，如下面的例子使用curl命令获取与16502530000对应的主叫号码：

```
$ curl https://api.opencnam.com/v2/phone/16502530000?format=pbx
  GOOGLE INC
```

- mod_cluechoo：该模块是一个例子模块，主要是教大家怎么写模块。另外，它还带了一个好玩的例子，如在命令行上执行cluechoo命令在屏幕上将会看到开过一个小火车。
- mod_commands：提供了大部分系统的命令API。
- mod_conference：多人语音及视频会议。
- mod_curl：使用libcurl作为一个HTTP客户端向Web服务器发送请求，也可以得到返回的结果。
- mod_db：该模块提供一组接口，用于使用API或App对数据库表进行增、删、改、查等操作。
- mod_directory：该模块用于按姓名呼叫用户。如果不知道用户的分机号，但知道用户的名字，则可以通过输入该用户名字的前几位进行拨叫。如在下面的例子中，我们可以通过拨打411进入directory程序。

```
<extension name="directory" continue="true">
  <condition field="destination_number" expression="^411$">
    <action application="directory" data="default $${domain} default"/>
  </condition>
</extension>
```

当系统提示输入名字时，我们输入9378可以找到Brian West（默认是按last_name查找的，West对应键盘按键9378）。

- mod_distributor：如果需要通过多个网关出局，则该模块可以帮助将呼叫根据一定的比例分发到不同的网关。
- mod_dptools：提供了大部分系统的App。
- mod_easyroute：用于根据号码路由，对大规模的DID比较有用。
- mod_enum：通过ENUM查询可以根据E164号码找到用户的SIP地址。该模块也提供了一个enum Dialplan。
- mod_esf：esf是Extended SIP Functionality的缩写，即扩展的SIP功能。它提供通过Muticast方法实现组拨的功能。
- mod_esl：该模块用于实现两个FreeSWITCH间的ESL对接，即一个FreeSWITCH可以作为另一个FreeSWITCH的ESL客户端访问它。
- mod_expr：提供expr表达式计算。
- mod_fifo：一个先入先出队列（First In First Out），可以用于简单地呼叫中心排队。
- mod_fsk：收发FSK（Frequency-shift keying，移频键控）信息。
- mod_fsv：fsv是FreeSWITCH Video的缩写，它使用了一种私有的格式来进行视频录像，可以支持任何编码的视频格式。它只存储原始的RTP包，对视频流不进行任何

处理。

- mod_hash：用于操作系统内部的哈希表。可以存储一些简单的数据。
- mod_httapi：一种 HTTP 格式的 API 接口，可以通过 HTTP 方式写 IVR。
- mod_http_cache：通过 HTTP 方式上传和下载文件，并可以进行本地缓存。
- mod_ladspa：使用 ladspa 库对声音进行处理，可以让声音更好听。
- mod_lcr：LCR 是 Least Cost Routing 的缩写，即最省钱的路由。它会根据数据库配置的路由信息和费率找到最省钱的路由。
- mod_limit：用于系统资源限制。
- mod_memcache：与 Memcache 交互，类似于把 mod_hash 的数据存到远程的 Memcache 中。
- mod_mongo：与 Mongodb 交互，类似 mod_memcache。
- mod_mp4：提供 MP4 文件的播放支持。
- mod_nibblebill：一些简单的计费功能，可用于预付费和电话卡类的应用。
- mod_oreka：使用 oreka 进行录音。oreka 是一款开源的录音软件。
- mod_osp：通过 Open Settlement Protocol 查找路由或上报 CDR。
- mod_rad_auth：使用 radius 服务器进行鉴权。
- mod_random：通过访问 /dev/hwrandom 设备影响随机数的熵。
- mod_redis：与 redis 服务器交互，类似于 mod_memcache。
- mod_rss：访问 RSS（Really Simple Syndication，简易信息聚合）数据。
- mod_skel：一个模块的例子框架。
- mod_sms：处理文本消息，如收发 SIP MESSAGE 消息等。它实现了消息路由（Chazplan），类似于 Dialplan。
- mod_snapshot：可以截取一段声音的快照。
- mod_snipe_hunt：一个简单的例子模块。
- mod_snom：用于 snom 话机的一些特性。
- mod_sonar：该模块类似于一个真正的声呐。首先你可以在远端启动一个服务器，其能对来话执行 echo App。然后在本地的 FreeSWITCH 上产生一些铃音，发送到远端的服务器上再反射回来，然后使用 VAD 检测功能可以检测这些铃音，从而可以在某种程度上确定网络的质量。
- mod_soundtouch：使用 soundtouch 库对声音进行处理，可以增加音效。
- mod_spandsp：使用 spandsp [⊖]支持一些语音编码及传真功能。
- mod_spy：用于监视某个话机，当该话机有通话时，本机振铃同时可以监听。
- mod_stress：使用快速傅里叶变换（Fast Fourier Transform，FFT）检测重音。
- mod_translate：通过既定的规则对号码进行翻译。
- mod_valet_parking：电话停靠。类似于泊车，有来电时可以将来电依靠在某个泊位

⊖ http://www.soft-switch.org/。

上，然后通知某人拨打指定的号码将来话“接”走。

- mod_vmd：提供 Voicemail 声音检测。在国外，好多电话都有自动应答功能，如“您好，主人不在家，请留言”。使用该模块可检测到这种声音，应用程序在自动外呼时就可以根据它的结果判断是人工接听的还是机器接听的。
- mod_voicemail：语音邮箱。
- mod_voicemail_ivr：带 IVR 导航的语音邮箱。

2. asr_tts

该目录下的模块提供自动语音识别及语音合成的功能。

- mod_cepstral：使用 Cepstra 语音库支持 TTS。
- mod_flite：使用 Festival Lite 库支持 TTS。
- mod_pocketsphinx：使用 pocketsphinx 库支持语音识别。
- mod_tts_commandline：通过命令行程序使用 TTS。
- mod_unimrcp：通过 uniMRCP 协议与其他 ASR/TTS 产品对接。uniMRCP 是一个标准的协议，很多语音产品都支持它。

3. codecs

该目录下的模块各种类型的音频、视频编码。大多数名称都很直观，这里不再过多解释。

- mod_amr。
- mod_amrwb。
- mod_b64：Base64 编码，可以传输任意数据。
- mod_bv：BroadVoice 的编码。
- mod_celt。
- mod_codec2。
- mod_com_g729：商业的 G.729 编码，可转码，需要许可证。
- mod_dahdi_codec：通过 DAHDI 库提供的编码。
- mod_g723_1。
- mod_g729：只支持透传。
- mod_h26x：提供 H261，H263，H264 等视频编码。
- mod_ilbc。
- mod_isac。
- mod_mp4v。
- mod_opus。
- mod_sangoma_codec：通过硬件板卡支持包括 G.729、iLBC 等多种编码。
- mod_silk。
- mod_siren。

- mod_skel_codec：例子模块。
- mod_speex。
- mod_theora。
- mod_voipcodecs。
- mod_vp8。

4. dialplans

该目录下的模块提供拨号计划。

- mod_dialplan_asterisk：类似于 Asterisk 格式的拨号计划。
- mod_dialplan_directory：通过 LDAP 查询拨号计划。
- mod_dialplan_xml：XML 拨号计划。

5. directories

该目录下的模块提供目录服务。

mod_ldap：通过 LDAP 提供目录服务。

6. endpoints

该目录下的模块提供各种 Endpoint 的实现。

- mod_alsa：使用 ALSA 声卡。
- mod_dingaling：连接 Google Talk。
- mod_gsmopen：使用无线上网卡上的 GSM 接口或使用手机上的 GSM 接口给外界发短信或与外界通话。
- mod_h323：连接 H.323 设备，使用 OpenH323 库实现。
- mod_khomp：使用 KHOMP 板卡。
- mod_loopback：提供 loopback 回环接口。
- mod_opal：连接 H.323 设备，使用 OPAL 库实现。
- mod_portaudio：通过 Portaudio 库支持本地声卡。
- mod_rtmp：通过 Adobe 的 RTMP 协议与浏览器中的 Flash 电话进行通话。
- mod_skinny：支持思科的 SCCP 协议话机。
- mod_skypopen：与 Skype 互通。
- mod_sofia：SIP 模块。

7. event_handlers

该目录下的模块提供事件处理功能。

- mod_cdr_csv：CSV 格式的话单。
- mod_cdr_mongodb：将话单写入 Mongodb。
- mod_cdr_pg_csv：将话单写入 PostgreSQL 数据库。
- mod_cdr_sqlite：将话单写入 SQLite 数据库。

- ❑ mod_erlang_event：对接 Erlang 节点，提供事件、日志、命令接口等。
- ❑ mod_event_multicast：将事件通过组播方式发出去。
- ❑ mod_event_socket：通过 ESL 库与第三方的接口。
- ❑ mod_event_test：测试。
- ❑ mod_event_zmq：使用 ZeroMQ 协议与第三方对接。
- ❑ mod_json_cdr：JSON 格式的 CDR。
- ❑ mod_radius_cdr：将 CDR 写入 Radius 服务器。
- ❑ mod_rayo：Rayo 支持。
- ❑ mod_snmp：SNMP 网管接口。

8. formats

该目录下的模块提供各种格式的文件。

- ❑ mod_local_stream：从本地文件生成媒体流。
- ❑ mod_native_file：支持原生文件读写，如直接读写 .PCMU 或 .G729 格式的文件。
- ❑ mod_portaudio_stream：使用 portaudio 库从本地声卡生成媒体流。
- ❑ mod_shell_stream：从 Shell 命令中生成媒体流。
- ❑ mod_shout：提 MP3 文件格式支持和远程 Shoutcast 服务器支持。
- ❑ mod_sndfile：使用 libsndfile 支持大多数的声音文件。
- ❑ mod_ssml：SSML 格式的文件支持。
- ❑ mod_tone_stream：生成铃流音。
- ❑ mod_vlc：使用 libvlc 提供媒体文件格式的支持。

9. languages

该目录下的模块提供各种嵌入式编码语言接口。

- ❑ mod_java：Java 的接口。
- ❑ mod_lua：Lua 的接口。
- ❑ mod_managed：微软平台的语言接口，如 C#、VB.NET 等。
- ❑ mod_perl：Perl 的接口。
- ❑ mod_python：Python 的接口。
- ❑ mod_spidermonkey：JavaScript 的接口。
- ❑ mod_v8：使用 V8 Javascript 引擎实现的 JavaScript 模块，可替换 mod_spidermonkey。
- ❑ mod_yaml：Yaml 的接口。

10. loggers

该目录下的模块提供与日志相关的功能。

- ❑ mod_console：控制台日志。
- ❑ mod_logfile：日志文件。
- ❑ mod_syslog：将日志写到 Syslog。

11. say

该目录下的模块提供多语种接口。

- mod_say_de：德语。
- mod_say_en：英语。
- mod_say_es：西班牙语。
- mod_say_fa：波斯语。
- mod_say_fr：法语。
- mod_say_he：希伯来语。
- mod_say_hr：克罗地亚语。
- mod_say_hu：匈牙利语。
- mod_say_it：意大利语。
- mod_say_ja：日语。
- mod_say_nl：荷兰语。
- mod_say_pl：波兰语。
- mod_say_pt：葡萄牙语。
- mod_say_ru：俄语。
- mod_say_th：泰语。
- mod_say_zh：汉语。

12. timers

该目录下的模块提供与定时器相关的功能。

- mod_posix_timer：Posix 定时器。
- mod_timerfd：使用 Linux 内核中的 timerfd 定时器。

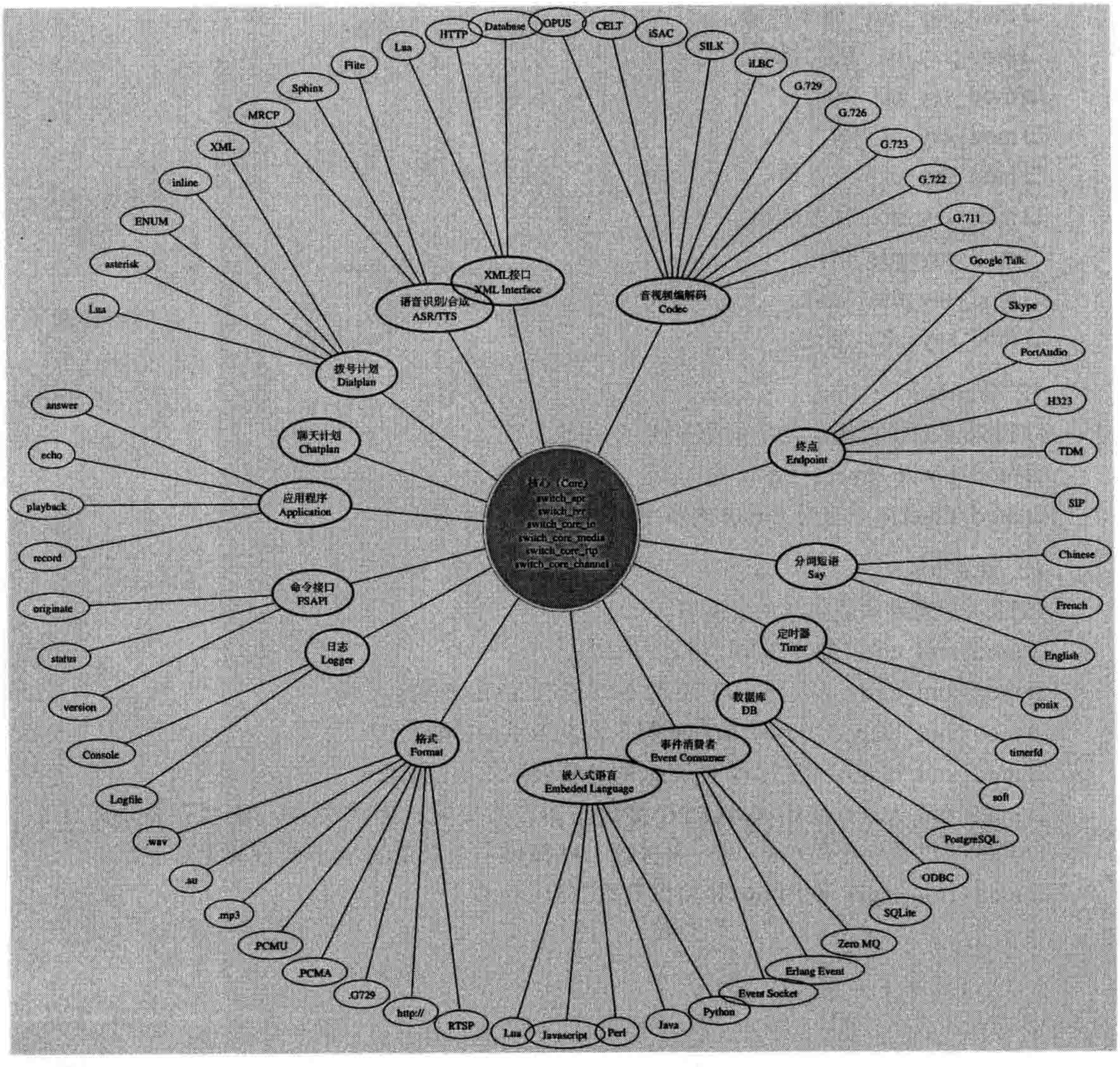
13. xml_int

该目录下的模块提供 XML 接口。

- mod_xml_cdr：使用 XML 格式写 CDR。
- mod_xml_curl：从远程 HTTP 服务器获取 XML 配置。
- mod_xml_ldap：从远程 LDAP 服务器获取 XML 配置。
- mod_xml_radius：从远程 Radius 服务器获取 XML 配置。
- mod_xml_rpc：使用 XMLRPC 接口与第三方交互，提供命令、日志及事件接口等。其本身也是一个简单的 Web 服务器，并提供一个简单的 Web 管理界面。
- mod_xml_scgi：使用 SCGI 协议获取 XML 配置。

附录 D

FreeSWITCH 架构详细示意图



后　　记

冬去春来，寒来暑往，岁月总是在慢慢地流逝着，不管我们是闲是忙。从决定写书到现在，算起来已经有将近四个年头了。

最初，我听取同事方舟的建议，将本书的内容以博客连载的形式使用“知识共享协议”（Creative Commons，CC）在网上发布，以便慢慢积累，听取大家的建议和反馈，并不断改进。

前几章放到网上以后，得到了很好的反馈。大家不仅很细心地帮我找错别字，还时不时地鼓动我出书，好多人都说书出了以后必定第一个购买。我不记得他们的名字，但是我很感动。可出书毕竟不是那么简单的事。首先，写书需要耗费大量的时间；其次，当时担心该书的读者群比较小，因此觉得这个想法离现实还比较远。

2012 年 6 月，我们在北京筹办了 FreeSWITCH 中文社区第一届开发者沙龙，也就是个小型的网友见面会。就在沙龙开始之前的几天，Tim Yang 先生资助我把当时手里的资料以书的形式印了出来，并在沙龙上作为奖品散发了一下。虽然时间仓促，排版比较粗糙而且错别字很多，但大家的反响还不错。而且由于这也算是仅有的 FreeSWITCH 中文资料，因此很多人都告诉我想要一本。当时书的名字叫《FreeSWITCH：VoIP 实战》。

后来，随着大家对本书需求的日益迫切，听取了中国呼叫中心行业资深评论家刘光大老师的建议和安排，方舟先生帮我联系了出版社，本书的写作就正式排上日程了。我随后也在当年（2013）年的 FreeSWITCH 开发者沙龙上宣布了本书的写作计划。最初，考虑到本书已初具规模，我想可能很快就能出版并跟读者见面，故打算在 2013 年 8 月份去美国参加 ClueCon 之前定稿。但在后来的编辑过程中却发现，自己印一本小册子和真正写书相比差距不是一般的大，甚至可以说根本就不是一回事。

首先，出版社的编辑检查得很仔细，遣词造句要流畅那自然不必说了，重要的是内容必须充实、有序、有理有据。由于考虑到本书是国内关于 FreeSWITCH 的第一本书，我们在敲定本书内容和目录时涵盖的内容就比较多，这就注定了需要更多的时间。其次是我工作很忙，能抽出来的完整时间不多，甚至有一段时间出差特别多，简直成了“空中飞人”。再加上，在这期间，由于前面的章节历史比较久远，而 FreeSWITCH 又更新太快，因此有些内容不得不重写，有的甚至重写了好几遍。但无论如何，我还是坚持加了一些夜班，并充分利

用了乘火车和飞机的时间，最终把本书完成了。

“完成”是一件令人欣慰的事，可令我比较为难的是，当书终于写成了的时候发现书又太“厚”了。因而，又不得不做了一些“精简”。

回想起来，写作的过程虽然有些艰辛，但写着写着也就过去了。而且写作的过程本身也是我学习的过程。为了照顾本书的各种背景、各种水平的读者，为了让读者更多地了解FreeSWITCH的方方面面，在本书的内容上，我尽量选择了不同语言、不同场景，并且涉及各种不同的技术、不同实现方法的例子，力求使大家通过本书对FreeSWITCH有一个全面而又不乏深度的认识。诚然，我并不是对FreeSWITCH中的每个方面都了解得很透彻，因而我也花了许多时间进行了大量的研究。在研究的过程中我还发现并修复了代码中许多的Bug，还对某些功能进行了优化和增强。每遇到不明白的问题，我便通过IM或电子邮件向Anthony询问，Anthony也总是不吝详细指导。慢慢地，我更加深入地了解了更多的FreeSWITCH内部的东西，间或也听到一些有趣的故事。除此之外，一年一度的ClueCon开发者大会也给了我与全世界FreeSWITCH开发者和使用者绝佳的面对面交流及学习的机会。

FreeSWITCH不仅是一个软交换系统，对各种编程语言和各种新技术和协议的支持俨然使用它成为了一个众多系统的黏合剂。在这个国内电信政策日益宽松和明朗，并强调个性化、强调创新的年代，相信它必大有作为。

FreeSWITCH的开发一向比较活跃，FreeSWITCH社区的人们也都很友好，这也是FreeSWITCH能够长期发展并走向成功的很重要的一个方面。最近几天，FreeSWITCH开发者们正在集中精力解决FreeSWITCH Bug跟踪系统上的问题，我也偶尔参与其中解决几个Bug。看来，真正的1.4版很快就要发布了。也许你拿到本书时，FreeSWITCH 1.4已经发布好几个版本了。最后，希望大家都加入我们的FreeSWITCH中文社区，多多交流技术和想法，为FreeSWITCH在全中国乃至全世界的壮大和发展作出贡献。

是为后记，与诸君共勉！

杜金房
于烟台

PDF电子书说明：

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ: 461573687, 或者 QQ: 2404062482。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！